VOLUME 06 | ISSUE 05

digit FastTrack
YOUR HANDY GUIDE TO EVERYDAY TECHNOLOGY

TO ANDROID SDK

- Android as a **development platform**
- Accessing **location-based services**
- **Alerting users** via notifications
- **Background services**
- **Telephony** and **SMS**
- **Android overview**
- User **Interface**
- **Wi-Fi**

A 9.9 Media Publication

www.
thinkdigit/forum

Join the forum to express your views and resolve your differences in a more civilised way.

thinkdigit
FORUM

Post your queries and get instant answers to all your technology related questions

one of the most active online technology forum not only in India but world-wide

JOIN NOW

www.thinkdigit.com

Fast Track

to

ANDROID SDK

Credits

The People Behind This Book

Editorial

Editor	Robert Sovereign-Smith
Head-Copy Desk	Nash David
Writers	Rahil Banthia, Nitish Varma

Design and Layout

Senior Designers	Baiju NV, Chander Dange, Vinod Shinde
Cover Design	Shigil N

May 2011

Free with Digit. Not to be sold separately. If you have paid separately for this book, please email the editor at editor@thinkdigit.com along with details of location of purchase, for appropriate action.

Contents

1 Android Overview 06

2 Android as a Development Platform 08

3 User Interface 15

4 Accessing Location-Based Services 36

5 Background Services 40

6 Alerting Users via Notifications 55

7 Telephony and SMS 60

8 Wi-Fi 93

Introduction

This Fast Track gets you started with the Android Software Development Kit (SDK) and helps you understand the ecosystem from a business perspective. You will be introduced to Android as a development platform and will learn to set up your environment for Android application development. We will cover some of the best practices in setting up your workspace in addition to listing out various steps for installation, compilation, testing and deployment.

To create projects and show the various codes and their significance in application development, we'll be using the Android Development Tools (ADT) plug-in for Eclipse IDE. You will get an insight into the workflow of application development, and be able to test-run it on Android Virtual Device (AVD, an emulator) or on a physical device (such as an unlocked Google Nexus S developer phone). This guide also runs you through some basic executable programs, and as is done in any programming tutorial, we start off with creating a 'Hello World!' project and compiling it to run on an emulator.

Once you're well versed with basic workspace components used for application development, you'll be shown how to create layouts and views to produce compelling user interfaces. We'll acquaint you with the basics of coding in Java and XML along with a listing of essential Widget properties. You'll also learn to log in to Android using LogCat. Just like in any computer program, you need to thread your process to create efficient and responsive programs. We'll introduce you to the basics of multi-threading in Android applications. The user interface section will teach you to optimize your layout and check the running process using Hierarchy Viewer. We'll learn to implement the same for a Twitter Status Update application.

Since your application will be deployed on a portable device capable of pin-pointing your location through network-based GPS,

we'll reveal how to access location-based services and implement geo-tags in your application.

Some applications run as services in the background without a user interface. You'll learn to create and control such services, and also set up their self-termination. You'll then be able to bind activities to services and set requisite priorities for these services. Threading these background services will follow. We'll also introduce you to the Toast view in Android, which is like a status update of a service on the user interface, and takes up minimal screen estate.

You'll see how easy it is to implement user notifications for hardware and software, which are similar to notifications of new text messages through flashing LEDs. We'll teach you to implement icons with notification so that it's an activity-specific indicator. Bluetooth services handling will also be covered.

Having understood the 'smart' of your smartphone, the next logical step would be telephony and SMS handling. In this section, you'll be familiarised with implementing 'phone' related activities and handling them. Since a user will connect to the internet using Wi-Fi at several instances, you'll learn to monitor, scan and create Wi-Fi networks and network configurations.

1 Android Overview

Android is a software stack for portable devices such as mobiles and tablets. The Android operating system is based on a modified Linux kernel. The Android developer community extends the functionality of the devices beyond the stock applications, and there already are over a hundred thousand Android apps in the Android Market (Google's online Android App store). Coding for these applications is mainly done in Java using Google-developed Java libraries. The Android platform is coded in C for its core, C++ for third-party libraries and Java for the user interface. Most of the code is under the Apache license, as free and open source software. The latest stable release of Android is the Honeycomb (Android 3.0.1) for tablets and Gingerbread (Android 2.3.3) for mobiles. It supports ARM, MIPS, Power and x86 platforms.

Android's open source stack runs on a Java-based, object-oriented application framework. It operates over Java core libraries running on the Dalvik virtual machine (VM). Prior to execution, Android applications are converted into Dalvik executables (DEX) format, rendering them suitable for portable devices with memory and processing speed constraints; as it is a register based architecture, unlike the stack machines of Java VMs. The database management system used is SQLite. Android uses Open Graphics Library for Embedded Systems (OpenGL ES) 2.0 3D graphics application programming interface (API).

1.1 Features

The platform is adaptable to large VGA displays, 2D graphics library and 3D library based on OpenGL. It supports all ranges of connectivity technologies such as: GSM/EDGE, CDMA, EV-DO, UMTS, Bluetooth, Wi-FI and WiMAX. The stock web browser is based on the open source WebKit layout engine coupled with Chrome's V8 JavaScript engine to render web pages. Android supports several media formats such as H.264, MPEG-4, AMR, AAC, MP3, WAV, Ogg, JPEG, PNG and GIF. It supports camera (video and still), touch screen, GPS, accelerometers, gyroscopes, magnetometers, proximity and pressure sensors, thermometers and accelerated 3D graphics; giving an application developer several options to mull over while developing compelling apps.

The software development kit (SDK) comprises an emulator (the Android Virtual Device), debugging tools, memory and performance profiling.

1.1.1 Comprehensive

Android is a comprehensive platform, i.e. it has the complete software stack to run on a portable device. The SDK consists of all required tools and framework for developers to write and deploy efficient applications for a device.

1.1.2 Made for Mobile Phones

Android was built with the idea of being incorporated in predominantly battery-powered smaller-sized devices restricted in memory and processing speed; thus delivering to a platform with a unique and better user experience. It doesn't assume a device's screen size, resolution, chipset, etc., and its core is designed for portability.

1.2 Incentive for Google

You may wonder how Google makes money if Android is released under a free and open-source license. The ideology that Google adopted while acquiring Android Inc in 2005 was to proliferate Android devices in the market, and not stop at launching just the gPhone. It wanted to develop a platform that would lure multiple manufacturers to imbibe this in their handsets. Predominantly being a media and advertising company, Google aspires to provide additional services to its advertisers through handheld devices.

Did you know? The first version of Android SDK was released without an actual device in the market. You don't need a phone for Android development; the SDK provides you with all the bits you need for developing on this platform.

Android, as such, is owned by the Open Handset Alliance – a non-profit formed by major mobile operators, device manufacturers and carriers and led by Google. It is committed to openness and innovation for mobile user experience.

1.3 Versions

Android has released several versions as listed below:

Version numbers change every time there is a considerable amount of bug fixes for an issue or when the API is revised.

Android Version	API Level	Codename
Android 1.0	1	
Android 1.1	2	
Android 1.5	3	Cupcake
Android 1.6	4	Donut
Android 2.0	5	Eclair
Android 2.01	6	Eclair
Android 2.1	7	Eclair
Android 2.2	8	Froyo
Android 2.3	9	Gingerbread
Android 2.3.3	10	Gingerbread
Android 3	11	Honeycomb

2 Android as a development platform

Applications for Android are developed using a group of tools provided with the Android SDK (software development kit). These tools can be accessed through an Eclipse plugin called Android Development Tools (ADT), or also from the command line. Developing with Eclipse is preferred as it has the ability to directly invoke tools required to develop applications. You can, however, choose any other text editor or an integrated development environment (IDE), and invoke the tools from the command line or use scripts. For the purpose of this Fast Track, we'll be referring to Eclipse IDE only.

Basic steps for developing applications:

1. Install Eclipse along with the ADT plugin.
2. Create Android Virtual Devices (AVD) or connect hardware devices you want to install your apps on.
3. Build your application and run it.
4. Debug your application using the debugging and logging tools available in the Android SDK.
5. Test your application on your hardware or on an emulator.

For references regarding tools to use if you are working on command line, check: http://developer.android.com/guide/developing/tools/index.html

2.1 Installing the SDK

This section will describe the basic know-how to prepare your computer for developing Android applications.

2.1.1 Getting your system ready

Check your system requirements:

- Windows XP (32-bit) or higher
- Mac OS X 10.5.8 or later (x86 only)
- Linux (tested on Ubuntu Linux, Lucid Lynx)
- GNU C Library (glibc) 2.7 or later is required
- On Ubuntu, version 8.04 or later is required
- 64-bit distributions must be capable of running 32-bit applications

Install the Java Development Kit (JDK) from http://java.sun.com/javase/downloads/index.jsp.

Download Eclipse 3.5 (Galileo) or higher from http://www.eclipse.org/downloads/

2.1.2 Downloading SDK Starter package

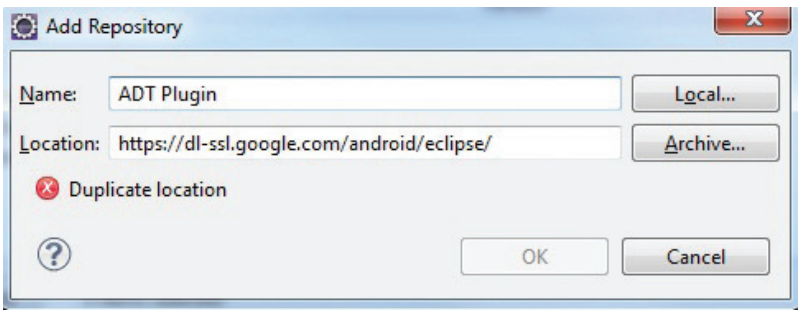
The SDK starter package is not a development environment, but includes only the core SDK tools. Download SDK from http://developer.android.com/sdk/index.html and run the installer to a known location (or unzip the file if you downloaded .zip or .tgz package).

2.1.3 Installing ADT plugin for Eclipse

Android Development Tools (ADT) is a custom plugin for the Eclipse IDE: a powerful environment to develop Android applications. Complemented by Eclipse, the ADT lets you quickly set up Android projects, create a compelling user interface, debug using Android SDK tools and export APKs (Android Packages) to distribute your application. It's the fastest way to get started with Android. Use the update manager feature of Eclipse to download and install the ADT plugin:

1. Select Install New Software ... from Help menu of Eclipse.
2. Click Add on the top right corner, and enter the following and click OK.

- Name : ADT Plugin
- Location : https://dl-ssl.google.com/android/eclipse/
- It'll display 'Duplicate location' if the URL is already added in your list of repositories.



Adding Repository

3. In the Available Software view, you should now see Developer Tools added to the list. Select the checkbox next to Developer Tools, which will automatically select the nested tools Android DDMS, Android Development Tools, Android Hierarchy Viewer and Android Traceview. Click Next.
4. In the resulting Install Details dialog, the Android DDMS and Android Development Tools features are listed. Click Next to read and accept the license agreement and install any dependencies, then click Finish.
5. Restart Eclipse.

2.1.4 Configure ADT Plugin

After a successful download, modify your ADT preferences to point to the SDK directory.

1. Select Window > Preferences...
2. Select Android from panel on the left of the dialog box
3. Enter location of the SDK directory in SDK location
4. Click Apply and Ok.

You can update the plugin by selecting 'Check for Updates' option from Help.

2.1.5 Adding platforms and components

Use the Android SDK and AVD Manager (included in the SDK starter package) to download essential SDK components into Eclipse. The SDK separates major parts of SDK—Android platform versions, add-ons, tools, samples, and documentation—into a set of individually installable components. The SDK starter package downloaded above includes only the latest version of the SDK Tools, hence you need to download at least one Android platform and the SDK Platform-tools (tools that the latest platform depends upon). Launch the Android SDK and AVD Manager from the Window menu of Eclipse and download packages using the graphical UI.

It is recommended to install at least SDK tools, SDK platform tools and SDK Platform from this manager. Documentations, Samples and USB Drivers can be useful. Advanced users and users who plan to publish their applications on the Android Marketplace should download Google APIs and Additional SDK platforms as well.

2.2 Testing the Installation

To ensure that the installation was successful, we start with a simple "Hello World" program.

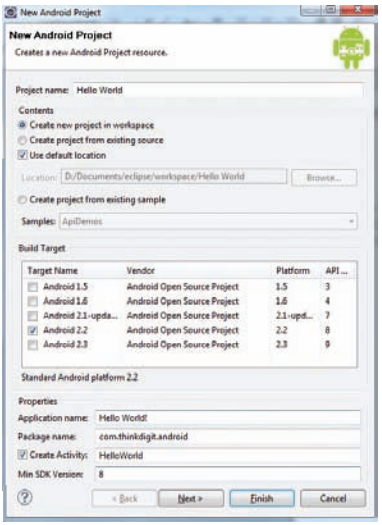
2.2.1 Creating a New Project

Choose File > New > Android Project from your Eclipse menu. If Android Project is not present, locate it from 'Other' option in New. Fill the following details in the new project dialog window:

- **Project name:** is an Eclipse construct where Eclipse organizes everything into projects. Project name should be one word, here we type 'HelloWorld'
- **Build target:** It tells the build tools which version of the Android platform you are building. Here you'll see a list of available platforms

and add-ons you installed along with your SDK. Choose one of the Android add-ons (like 'Android 2.2' for this program)

- **Project Properties:** The application name will be any English name you want to give your application, say 'Hello World!'
- **Package name:** is a Java construct, where all source code is organized into packages. Packages are important as they specify visibility of objects between the various Java classes. In Android, packages play a pivotal role for application signing purposes. A package name should be the reverse of your domain name, for instance 'com.thinkdigit.android'
- **Create Activity:** Activities correspond to various screens of your application. It is also represented by a Java class, hence you should adhere to Java class naming conventions, i.e. start with an upper case letter, and use upper case letters to separate words (no spaces). Here our Activity name will be 'HelloWorld'.
- **Minimum SDK Version:** Minimum version of the SDK represented by API level, required to run the application. This number should be as low as possible to allow scalability of your application to as many users as possible. We keep it '8' here.



Hello World

Files automatically created by Eclipse

2.3 Manifest File

This XML file (AndroidManifest.xml) explains the contents of the application, its building blocks, its required permissions etc.

```
1 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
2 package="com.marakana" android:versionCode="1"
android:versionName="1.0">
3 <application android:icon="@drawable/icon" android:label="@
string/app_name">
4 <activity android:name=".HelloWorld" android:label="@string/
app_name">
5       <intent-filter>
6       <action android:name="android.intent.action.MAIN" />
7 <category android:name="android.intent.category.LAUNCHER" />
8 </intent-filter>
9 </activity>
10 </application>
11 <uses-sdk android:minSdkVersion="8" />
12 </manifest>
```

2.4 Layout File

This file (res/layout/main.xml) specifies the look of the screen, its containers and boxes. We have only one screen loaded with the HelloWorld.java code. Contents of res/layout/main.xml:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/
res/android"
android:orientation="vertical" android:layout_width="fill_parent"
android:layout_height="fill_parent">
3 <TextView android:layout_width="fill_parent"
android:layout_height="wrap_content" android:text="@string/hello" />
4 </LinearLayout>
```

2.5 Strings File

strings.xml in res/values contains all the text used by the application. The names of buttons, labels, default text and similar types of strings go into this file. It is a good practice to separate the concerns of various files. In this case, Layout XML is responsible for widget layout while Strings XML handles the textual content. Contents of res/values/strings.xml

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <resources>
3.      <string name="hello">Hello World, HelloWorld!</string>
4.      <string name="app_name">Hello, World!!!</string>
5. </resources>
```

2.6 R File

R file in the /gen folder connects resources to Java, and is automatically generated. It is recreated with every change in the res directory. Eclipse automatically handles this file.

2.7 Java source code

Java Source code drives everything. This is what gets converted into a Dalvik executable file to run your application. Your code looks like this:

```
1 package com.thinkdigit.android;
2 import android.app.Activity;
3 import android.os.Bundle;
4 public class HelloWorld extends Activity {
5 /** Called when the activity is first created. */
6 @Override
7 public void onCreate(Bundle savedInstanceState) {
8 super.onCreate(savedInstanceState);
9 }}
```

This is where you add your own bit of code for additional operations.

2.8 Emulator

An Emulator runs your application as a virtual layer on another host so that you don't have to spare an Android phone to test your apps. It runs the same base code as an actual device.

A true emulator based on QEMU (type of a processor emulator covered under GNU GPL version 2) is shipped along with Android SDK. It acts like a hosted virtual machine monitor. We'll now create an Android Virtual Device, or an AVD to use this emulator. Start the tool called Android SDK and AVD Manager from the Eclipse Window menu.

Clicking on New opens a 'Create new Android Virtual Device (AVD)' window. Specify the parameters for your new AVD.

- **Name:** Any English name you choose
- **Target:** Version of Android you want installed on this AVD. If you don't

have targets, download them from the 'Available Packages' window of Android SDK and AVD Manager.

- You can also specify some hardware properties in this dialog box.

Once done, you'll notice a new virtual device created on your AVD and SDK Manager. You can now start this AVD from the 'Start' option in the manager. This will open the emulator in a pop-up.

2.8.1 Emulator or device?

Even though running your application on an emulator is similar to that on a physical device, there are a few exceptions, e.g. visualizing the action of some sensors. This can be tested on a physical device by programming using the USB Debugging Mode.

2.8.2 Controlling the emulator

You interact with the device just like you would with a physical device. You use the mouse to give touch inputs and type on keyboard keys to trigger the simulated device keys. Some keyboard shortcuts:

2.8.3 Limitations

The emulator gives no support for placing or receiving actual phone calls. However, placed and received phone calls can be simulated through the emulator console. It doesn't support USB connection, camera input, headphone jack insert, determination of SD card insert or eject, or Bluetooth.

2.9 Hardware device

Before releasing to users, it's always better to test the application on a real device. You can use any Android device as the environment to run, debug and test the applications you create. SDK tools make installation of the application on the device simple on every compilation.

Set up the device for development by:

1. Declaring the application as Debuggable in the AndroidManifest.xml by adding android:debuggable="true" to the <application> block
2. Turn on USB debugging on the device from Applications > Development
3. Set up your system to detect the device. Windows users need to install drivers for the device from http://developer.android.com/sdk/oem-usb.html. Mac OS X will automatically detect it.

Your device is ready for Debugging, for which you can use the DDMS debugging utility (as described in the User Interface chapter).

3 User interface

This chapter covers the basic know-how for building an Android user interface. At the end of this chapter, you'll be able to create an activity and an XML layout for the activity, and you'll learn how to connect it to Java. We'll also cover Views (or Widgets) and Layouts, and learn how to handle Java events such as clicks. You'll also be able to add support to Twitter-like APIs through external *.jar files, which will enable your app to make calls to the cloud.

3.1 Creating a user interface

A user interface in Android can be created in two ways, viz. using XML or writing Java code to develop the UI.

3.1.1 Declarative approach

Declarative user interfaces can be created using XML to declare what the UI will look like. It's similar to creating a web page using HTML. You use tags and specify which elements will appear on your screen. The advantage of this tool is that you can use "What You See Is What You Get" (WYSIWYG) tools, most of which are shipped with the Eclipse Android Development Tools (ADT) Extension.

However, XML only gives you the freedom to easily declare the look and feel of your user interface; it doesn't handle the user input well.

3.1.2 Programmatic approach

The programmatic approach for developing the Android user interface involves coding in Java. Android is more or less similar to Java AWT or Java Swing development. Hence, in order to create a button, you declare a button variable, create its instance, add it to a container and set the button properties such as color, text, size and background. You may have to write a code to declare the action of a button on click or hold; i.e. it involves considerable amount of coding in Java.

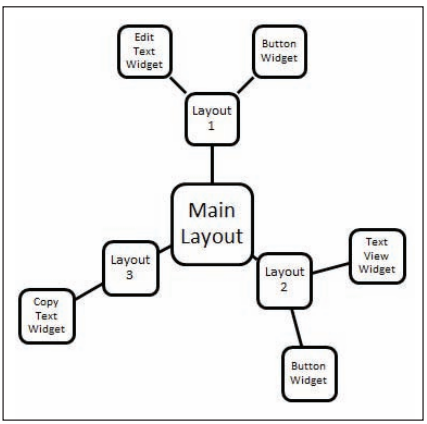
3.1.3 The right way

The best practice is to use the best of both the approaches. Use XML to declare all static elements of the user interface such as the layout, and the widgets etc. and switch to the programmatic approach to define the actions

of a user with the widgets in the user interface. Though the developer adopts this approach, all the XML code is blown up into Java code by your ADT.

3.2 Views and layouts

Android user interface elements are organized into layouts and views. View comprises everything you see, such as a button, label or text box. Layouts organize views; it covers grouping buttons and labels or groups of such elements. Layouts are like Java containers and Views are like Java components. Views are sometimes referred to as widgets (These are the views in the activities, and not the App Widgets embedded in the home screen application).



Layouts and Views relationship

You can embed layout within another giving a parent-child hierarchy, which can further contain the widgets.

3.2.1 LinearLayout

LinearLayout simply lays out its children horizontally or vertically next to each other. The order of the children in the layout matters, as in LinearLayout allocates space to each child in the order they are added. Hence, if most of the space is allocated to an older child, there'll be very little left for subsequent widgets in this layout. However, nesting multiple LinearLayouts proves heavy for your memory resources and kills your battery life.

3.2.2 TableLayout

This layout places the children in a tabular form. It consists of only other TableRow widgets. TableRow represents a row in a table. These can also contain other UI widgets. TableRow widgets are like LinearLayout with horizontal orientation, as they are laid out next to each other horizontally. TableLayout is similar to <table> element in HTML and TableRow is like <tr>.However, a number of columns are defined dynamically based on the number of views added to a table row.

3.2.3 FrameLayout

FrameLayout places its children like a deck of cards. The latest child covers the previous. This is useful for tabs on a layout; for instance, it's also used as a placeholder for other views, which can be added programmatically at a later time.

3.2.4 RelativeLayout

RelativeLayout, as the name suggests, lays out its children relative to each other. This position is based on the ID set for the particular child view. RelativeLayout doesn't require nesting of layouts to create a certain look, and minimizes the total number of widgets that need to be drawn. This improves the overall performance of the application. RelativeLayout positions a child relative to other children based on the ID set for the particular child view.

3.2.5 AbsoluteLayout

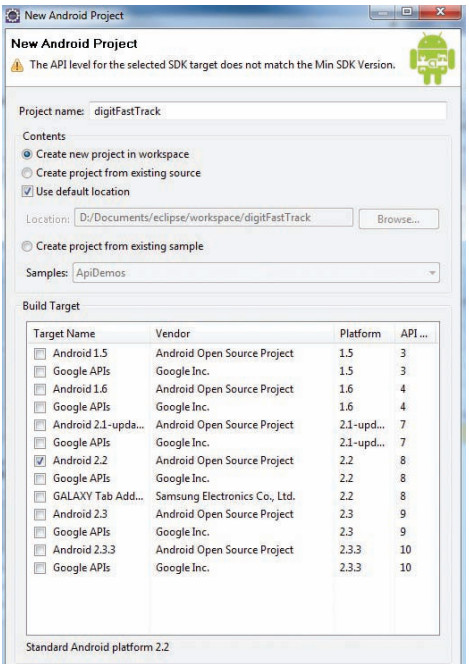
You can define absolute coordinates on the home screen to define the position of children using the AbsoluteLayout. It's a layout for WYSIWYG tools. However, the simplicity of its implementation and attractiveness come at the cost of flexibility. It doesn't adapt well to changes in size or orientation of the screen.

3.3 Getting Started

Here we'll start our project. Open Eclipse and click on File > New > Android Project.

- **Project Name:** Eclipse organizes your project by the name you provide here. Try not to use any spaces for easy access via command line at a later stage.
- **Contents:** Set to 'Create new project in workspace'.
- **Build Target:** Indicate the type of Android system you intend to run the application on.
- **Application name:** A text name for your app
- **Package name:** Adhere to Java package naming conventions for this:
  - Defined using hierarchical naming pattern
  - Levels in hierarchy separated by periods (.)
  - Packages lower in hierarchy are sub-packages of the immediate higher package
  - Naming convention avoids the possibility of two published packages with the same name. This allows for unique namespaces for packages widely distributed.

- Package name generally begins with top level domain name of the organization, followed by organization's domain and the other sub-domains, all listed in reverse order.
- Package names should be all lowercase wherever possible.
- Refer to Section 7.7 of Java Language Specifications (http://java.sun.com/docs/books/jls/) for further details on Package naming conventions.



New Project dialog box

- **Create Activity:** Adhere to Java class naming conventions to create an activity as a part of the project.
- **Min SDK Version:** This is the minimum version of Android SDK that must be installed on the device of the app user for it to run the application. However, choose the lowest possible API level if your app is scalable.

3.3.1 Designing the user interface

We'll now design a user interface for the screen which we'll use to enter a status and lay a button to update it. You'll find a file called main.xml in the res/layout folder. Rename it to status.xml by selecting main.xml and clicking Alt + Shift + R. Eclipse automatically looks up all the references to the file and updates those as well. However, this automatic feature only works well with renaming Java files, not with XML files. So, to rename these files, we need to change the line in Java where we refer to it via the R class.

Select StatusActivity.java from src/com/thinkdigit/envy folder. You'll see a code like this:

```
1 package com.thinkdigit.envy;
2 import android.app.Activity;
3 import android.os.Bundle;
4 publicclass StatusActivity extends Activity {
5 /** Called when the activity is first created. */
6 @Override
7 publicvoid onCreate(Bundle savedInstanceState) {
8 super.onCreate(savedInstanceState);
9 setContentView(R.layout.status);
10.}
11.  }
```

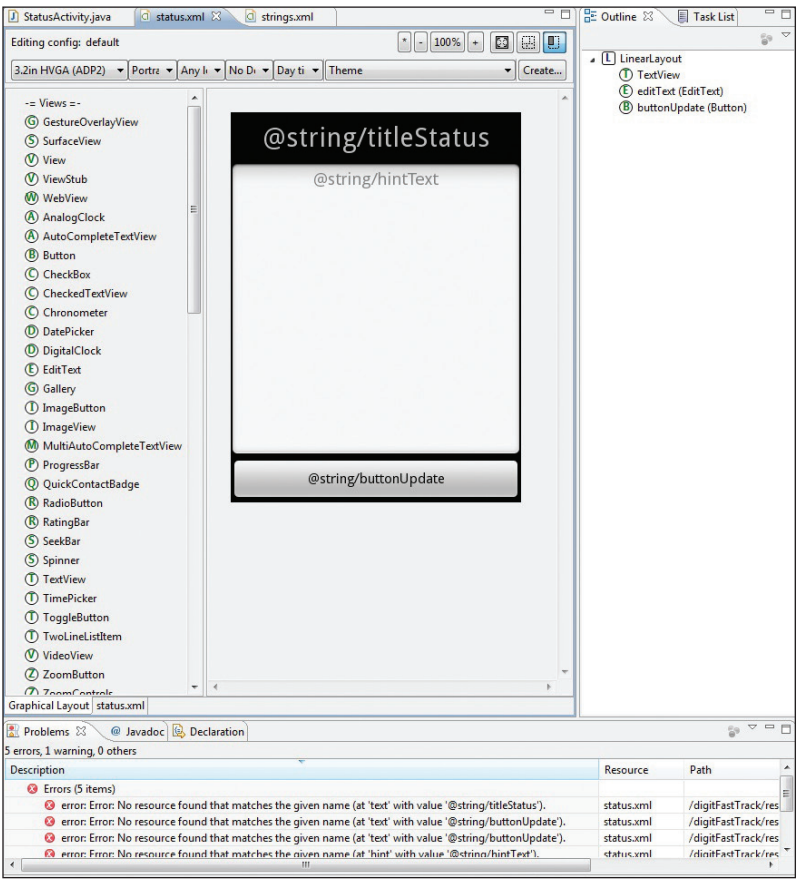
Change `R.layout.main` on line 9 to `R.layout.status`.

Now, double-click on status.xml in res/layout folder. You'll see a screen with four components: a title at the top, which is the TextView widget; a text area to type the 140 character status update, where we use an EditText widget; a button at the bottom to update the status, using the Button widget; and a layout containing all these widgets aligned in vertical fashion. Here we're using LinearLayout. This view is the Graphical layout mode.

You can view the raw XML code by clicking on the status.xml tab at the bottom of this screen. This code was generated in the Eclipse graphical layout. ADT Eclipse plug-in provides this to help you work with Android-specific files. Replace the code in status.xml with the following code:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <!--  Main Layout of StatusActivity -->
3 <LinearLayout xmlns:android="http://schemas.android.com/apk/
res/android"
4 android:orientation="vertical"
5 android:layout_width="fill_parent"
6 android:layout_height="fill_parent"
7 >
8 <!-- Title text view -->
9 <TextView
10 android:layout_width="fill_parent"
11 android:layout_height="wrap_content"
12 android:gravity="center"
13 android:textSize="30sp"
14 android:layout_margin="10dp"
15 android:text="@string/titleStatus"
```

```
16 />
17 <!-- Status EditText -->
18 <EditText
19 android:layout_width="fill_parent"
20 android:layout_height="fill_parent"
21 android:layout_weight="1"
22 android:hint="@string/hintText"
23 android:id="@+id/editText"
24 android:gravity="top|center_horizontal">
```



Graphical layout mode for status.xml

```
25 </EditText>
26 <!-- Update Button -->
27 <Button
28 android:layout_width="fill_parent"
29 android:layout_height="wrap_content"
30 android:text="@string/buttonUpdate"
31 android:textSize="15sp"
32 android:id="@+id/buttonUpdate">
33 </Button>
34 </LinearLayout>
```

As you can see in the Graphical layout, the title shows "@string/titleStatus" and so forth for the EditText and the Button widget (also shown as errors in the Problems). This is because the variables are not declared in the strings.xml file. To declare a string, open strings.xml from res/values folder and click on Add button, select String element from the dialog box that opens. Enter the attributes for the string as in the image below and save the strings.xml file (by pressing `[Ctrl] +[S]`).

3.3.2 Essential widget properties

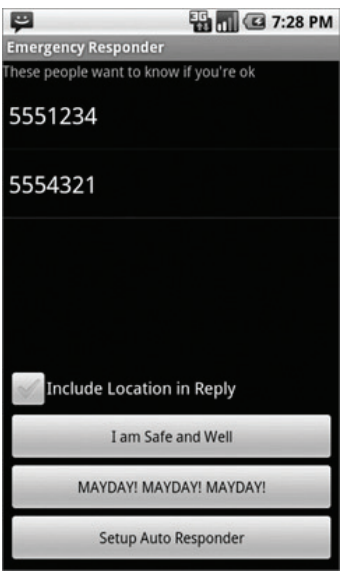
You may need to know a few details in the code, which we'll cover in this section. Properties most regularly used:

layout_height and layout_width

Used to define the space a widget asks from its parent layout. If you give absolute sizes in pixels or inches, the widget will be skewed on screen sizes it's not designed for. It's better to use relative sizes by implementing `fill_parent` (all available space from parent) or `wrap_content` (as much space as it needs to display own content) for the value. API level 8 and above use `match_parent` in place of `fill_parent`.

layout_weight

Layout weight is given a number between 0 and 1 which defines the weight of our layout requirements. For instance, if Status EditText is weighted 0 and

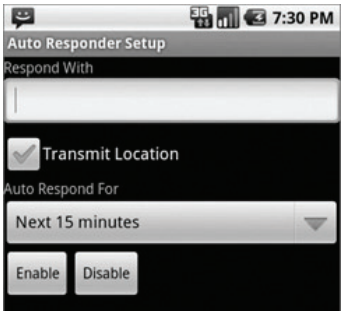


Layout with weight valued 0

requires layout height of fill_parent, it will push the Update button out of the screen, as request for space from Button followed request for Status. However, setting Status widget's weight to 1 allocates all available space in height, but yields to other widgets (like the Update button here) that may need space.

layout_gravity

This property specifies the position of a particular widget within its layout, horizontally and vertically. Values you can specify include top, center, left and so forth. But, if your widget is set to `fill_parent`, centering it won't do anything, as it's already taking all the available space. However, if the title `TextView` had its width set to `wrap_content`, centering with `layout_gravity` works.



Centering with layout_gravity

gravity

Gravity property specifies positioning of content within the widget itself. The choice between `layout_gravity` and gravity depends on the size of the widget and the desired look.

text

Widgets like Button, EditText and TextView have this property. It specifies the text for the widget. A good practice is to define all text in the strings.xml resource and refer to a particular string using @string/<String_name>, else the layout will work only in one locale or language.

id

This specifies a unique identifier for a widget in a particular layout resource. Use of id should be kept to a minimum to prevent clutter. Widgets that need to be manipulated in Java require an id. Id has a format: @id/<id_name>

3.4 Building with Java

You'll find a class named StatusActivity.java in src/com/thinkdigit/envy, which was created by the Eclipse 'New Project' dialog. It forms part of the package com.thinkdigit.envy

3.4.1 Application-specific Object and Initialization code

We start by subclassing a base class provided by the Android framework and overriding certain inherited methods. We subclass Android's activity

class and override the onCreate() method. Activities go through a certain state machine, which is its life cycle. The transition in state we want to bring is to override the onCreate() method invoked by ActivityManager of the system when the activity was created. The onCreate() method will set up the button to respond to clicks, and further connect it to the cloud. It takes a Bundle (small amount of data passed into an activity through the intent that triggered it) as a parameter, which is limited to basic data types; complex ones need to be encoded. To override a method, first call the original parent method, hence the super.onCreate() call. After subclassing the framework's class, override the appropriate method and call super's method in it. The code now does the same thing as the original class, but now we have a placeholder to add our own code.

We typically start by writing some Java code that opens up our XML layout file, parses it, and for every element in the XML file, it creates a corresponding Java object in the memory space. The code sets every attribute of an XML element in our Java object, i.e. inflates from XML are done by adding setContentView(R.layout.status);

R class is a set of automatically generated pointers which connects Java to XML and other resources in the /res folder. R.layout.status points to status.xml file. The setContentView() method reads the XML file, parses it, creates appropriate Java objects, sets object properties, sets up parent-child relationships, and in all inflates the entire view.

Not only user-created objects alone, but Android's user interface objects also define methods and respond to external stimulus.

Hence, to execute a Button click action, define an onClick() method and put the code you want to execute. In addition to this, you must run setOnClickListener method on the Button. This can be passed as an argument to the listener as the object is where onClick() is defined.

Your code for StatusActivity.java should look like:

```
1 package com.thinkdigit.envy;
2 import winterwell.jtwitter.Twitter;
3 import android.app.Activity;
4 import android.os.Bundle;
5 import android.util.Log;
6 import android.view.View;
7 import android.view.View.OnClickListener;
8 import android.widget.Button;
9 import android.widget.EditText;
```

```
10 publicclass StatusActivity extends Activity implements
   OnClickListener{
11 privatestaticfinal String TAG = "StatusActivity";
12 EditText editText;
13 Button updateButton;
14 Twitter twitter;
15 /** called when the activity is first created. */
16 @Override
17 publicvoid onCreate(Bundle savedInstanceState) {
18 super.onCreate(savedInstanceState);
19 setContentView(R.layout.status);
20 editText= (EditText) findViewById(R.id.editText);
21 updateButton=(Button) findViewById(R.id.buttonUpdate);
22 updateButton.setOnClickListener(this);
23 twitter = newTwitter("username", "password");
24 twitter.setAPIRootUrl("http://envy.thinkdigit.com/api");}
25 // Button Click
26 publicvoid onClick(View v){
27 twitter.setStatus(editText.getText().toString());
28 Log.d(TAG, "onClicked");
29 }}
```

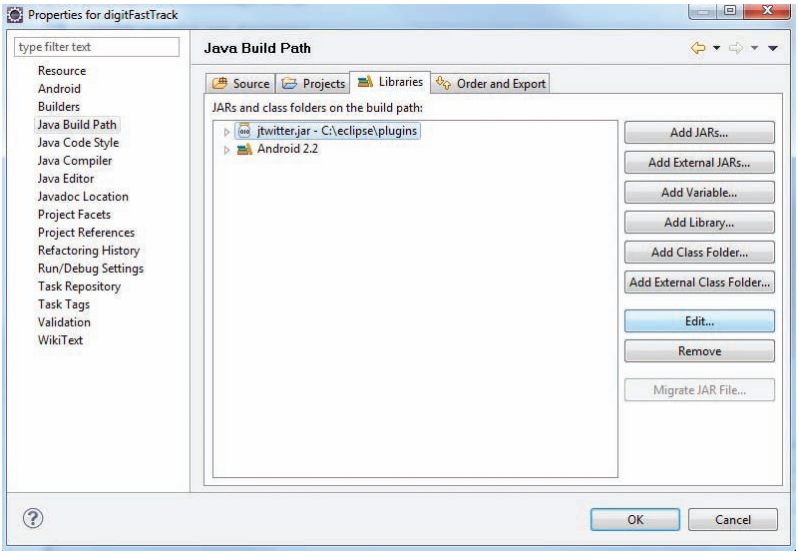
3.4.2 Compiling code

Eclipse builds every time you save your project, so remember to save your files after writing every block of code. Due to interdependency between Java and XML, transition from one file to another becomes more difficult with the current file broken, and makes it even more difficult to find errors. Clicking on the red 'x' marks generates possible solutions. This is like spell check in any word processor.

3.4.3 Adding libraries

We'll take a closer look at the jtwitter.jar library that lets us implement Twitter status updates. The connection happens through a series of web service calls. Winterwell Associates' jtwitter.jar library contains a simple Java class. This class interacts with online services, hence abstracting all the intricacies of network calls and data transfer.

Download the library from http://www.winterwell.com/software/jtwitter.php and insert it in your Eclipse project. Do this by clicking Alt + Enter with your



Properties for adding libraries

project selected in the Package Explorer. This opens the properties window. Navigate to Java Build Path on the left column, then to the Libraries tab. Here, click on 'Add External JARs...' and locate the downloaded jtwitter.jar library. Since Java searches for all the classes in the classpath, jtwitter.jar is added to the classpath by this method.

3.4.4 Internet Usage Permission

User must grant the right to access internet in order to make the application run. Android manages these permissions as a security measure. The user has to explicitly grant access to each application for internet usage. However, the user is not prompted again during application upgrades, unless the list of permissions changes. Open the AndroidManifest.xml file, which opens in the WYSIWYG editor with multiple tabs at the bottom. Click on the AndroidManifest.xml tab. This opens the XML view. Add <uses-permission android:name="android.permission.INTERNET"/> element within the manifest block. Your code should look like this:

1. `<?xml version="1.0" encoding="utf-8"?>`
2. `<manifest xmlns:android="http://schemas.android.com/apk/res/android"`

```
3. package="com.thinkdigit.envy"
4. android:versionCode="1"
5. android:versionName="1.0">
6. <application android:icon="@drawable/icon" android:label="@
   string/app_name">
7. <activity android:name=".StatusActivity"
8.      android:label="@string/app_name">
9. <intent-filter>
10. <action android:name="android.intent.action.MAIN" />
11. <category android:name="android.intent.category.LAUNCHER" />
12. </intent-filter>
13. </activity>
14. </application>
15. <uses-sdk android:minSdkVersion="4" />
16. <uses-permission android:name="android.permission.INTERNET" />
17. </manifest>
```

3.4.5 Logging in Android

Android offers a system-wide capability to log activities. In order to log, you can call the following anywhere in the code:

```
Log.d(TAG, message)
```

TAG and message are strings. TAG would be the name of your app, and a good practice is to define it as a Java constant for the entire class, such as:

```
private static final String TAG = "StatusActivity";
```

Log has different severity levels:

- `.d()` is for debug level
- `.e()` for error
- `.w()` for warning
- `.i()` for info
- `.wtf()` for errors that should never happen. (It stands for What a Terrible Failure)

Log messages are color-coded based on severity level.

3.5 LogCat

LogCat is a standard system-wide logging mechanism, where the Android system log is outputted. Logs can be easily viewed and filtered based on output. LogCat can be viewed in two ways: via Eclipse interface, or via the command line.

3.5.1 Eclipse Perspective

You can switch to the LogCat view in Eclipse by selecting the DDMS (Dalvik Debug Monitor Server) button on the top right corner of your Eclipse environment. Open it by selecting DDMS from Window > Open Perspective in the Eclipse menu. DDMS is the connection between the running application on the device and Eclipse. You can define filters for LogCat as well. Click on the little Green plus icon, which will open the filter dialog. This will open a separate window within the LogCat which will show results based on your filter parameters.

3.5.2 Command Line

Type the following command in your terminal window to view LogCat:

```
[user:~]> adb logcat
```

This gives the tail of the current LogCat and will keep updating as the device generates entries. For a list of command line syntaxes, visit: http://developer.android.com/guide/developing/tools/adb.html#logcatoptions.

3.6 Threading in Android

A Thread is a sequence of instructions executed concurrently. A CPU can process only one instruction at a time, but operating systems can interleave them on a single CPU. Based on thread priorities, the operating system allots time share to the threads. Built on the Linux platform, Android is in principle capable of running multiple threads simultaneously. However, a developer needs to know how applications use threads, in order to develop efficient applications.

3.6.1 Single Thread

Android applications run on a single thread by default, i.e. they run all commands serially. A successor is executed only upon completion of the current thread. Hence each call is blocking. The UI thread is responsible for drawing all elements on the screen, and also for processing all user inputs such as touch, pinch, clicks etc.

Running `StatusActivity` on a single thread creates problems during network calls to update status. The time for execution is not in our control, and the application cannot respond until the network call is completed, and Android may push to kill such a non-responsive thread; even if it's due to internet latency. This opens the Application Not Responding (ANR) dialog.

3.6.2 Multiple Threads

Running potentially long operations on separate thread is better implementation. The operating system slices the available processing time among multiple tasks running on multiple threads so that no task dominates execution. It may hence appear that the processes are occurring simultaneously. Hence loading the network call for updating on a separate thread will not block the UI thread, and the application will be much more responsive.

3.7 Accomplishing Multi-threading

Thread class in Java allows for multi-threading operations. Regular Java features can be used to put the network call in the background. A standard Java thread class doesn't allow one thread to update elements of the main UI thread, where we would need to synchronise with the current state. Android provides AsyncTask utility designed for this purpose.

3.7.1 AsyncTask

This Android mechanism helps to handle long operations that need to report to the UI thread. Create a new AsyncTask subclass and implement the following methods:

`doInBackground()` fills in for background activity

`onProgressUpdate()` instructs the follow-up activity upon certain progress

`onPostExecute()` defines the action upon process completion

Following will be your modified code in StatusActivity.java with implementation of an asynchronous thread:

```
1 package com.thinkdigit.envy;
2 import winterwell.jtwitter.Twitter;
3 import winterwell.jtwitter.TwitterException;
4 import android.app.Activity;
5 import android.os.AsyncTask;
6 import android.os.Bundle;
7 import android.util.Log;
8 import android.view.View;
9 import android.view.View.OnClickListener;
10 import android.widget.Button;
11 import android.widget.EditText;
12 import android.widget.Toast;
13  public class StatusActivity extends Activity implements
   OnClickListener{
```

```
14 private static final String TAG = "StatusActivity";
15 EditText editText;
16 Button updateButton;
17 Twitter twitter;
18 /** Called when the activity is first created. */
19 @Override
20 public void onCreate(Bundle savedInstanceState) {
21 super.onCreate(savedInstanceState);
22 setContentView(R.layout.status);
23 editText= (EditText) findViewById(R.id.editText);
24 updateButton=(Button) findViewById(R.id.buttonUpdate);
25 updateButton.setOnClickListener(this);
26 twitter = new Twitter("username", "password");
27 twitter.setAPIRootUrl("http://envy.thinkdigit.com/api");}
28 //Asynchronous Posting to Twitter
29 class PostToTwitter extends AsyncTask<String, Integer, String>{
30 //Call to initiate Background activity
31 @Override
32 protected String doInBackground(String... statuses){
33 try{
34 Twitter.Status status=twitter.updateStatus(statuses[0]);
35 return status.text;
36 }catch (TwitterException e){
37 Log.e(TAG, e.toString());
38 e.printStackTrace();
39 return "Failed to post";}}
40 //Call when there is a status to update
41 @Override
42 protected void onProgressUpdate(Integer... values){
43 super.onProgressUpdate();}
44 //Call upon completion of Background activity
45 @Override
46 protected void onPostExecute(String result){
47 Toast.makeText(StatusActivity.this, result, Toast.LENGTH_
   LONG).show();}}
48 // Button Click
49 public void onClick(View v){
50 String status = editText.getText().toString();
```

```
51 new PostToTwitter().execute(status);
52 Log.d(TAG, "onClicked");}}
```

Now, when the user clicks on the Update button, the activity will create a separate thread using AsyncTask and place the network operations on that thread. Upon completion, AsyncTask will update the main UI thread by popping up a toast message telling the user about the success or failure of the update. This makes the application more responsive and the ANR status is eliminated.

3.8 Other UI events

Till now we've seen the handling of click events. Now we'll show how to implement the following operations:

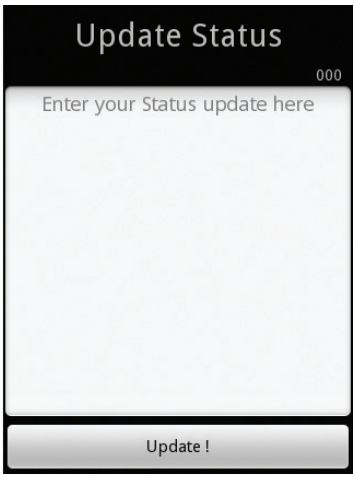
TextWatcher watches the changes in the text field. We'll use another TextView on our layout to indicate the number of characters available. We'll also implement a color shift from green to yellow to red as the user reaches the maximum limit of 140 characters.

The code will now look like the following:

3.8.1 Status.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <!--  Main Layout of StatusActivity -->
3  <LinearLayout  xmlns:android="http://schemas.android.com/apk/
   res/android"
4 android:orientation="vertical"
5 android:layout_width="fill_parent"
6 android:layout_height="fill_parent"
7 >
8 <!-- Title text view -->
9 <TextView
10 android:layout_width="fill_parent"
11 android:layout_height="wrap_content"
12 android:gravity="center"
13 android:textSize="30sp"
14 android:layout_margin="10dp"
15 android:text="@string/titleStatus"
16 />
17 <!-- Text Counter text view -->
18 <TextView
```

```
19 android:layout_width="wrap_content"
20 android:layout_height="wrap_content"
21 android:layout_gravity="right"
22 android:text="000"
23 android:id="@+id/textCounter"
24 android:layout_marginRight="10dp"/>
25 <!-- Status EditText -->
26 <EditText
27 android:layout_width="fill_parent"
28 android:layout_height="fill_parent"
29 android:layout_weight="1"
30 android:hint="@string/hintText"
31 android:id="@+id/editText"
32   android:gravity="top|center_
   horizontal">
33 </EditText>
34 <!-- Update Button -->
35 <Button
36    android:layout_width="fill_
   parent"
37   android:layout_height="wrap_
   content"
38       android:text="@string/
   buttonUpdate"
39 android:textSize="15sp"
40 android:id="@+id/buttonUpdate">
41 </Button>
42 </LinearLayout>
```



StatusActivity with counter

3.8.2 StatusActivity.java

```
1 package com.thinkdigit.envy;
2 import winterwell.jtwitter.Twitter;
3 import winterwell.jtwitter.TwitterException;
4 import android.app.Activity;
5 import android.graphics.Color;
6 import android.os.AsyncTask;
7 import android.text.Editable;
8 import android.text.TextWatcher;
```

```
9 import android.os.Bundle;
10 import android.util.Log;
11 import android.view.View;
12 import android.view.View.OnClickListener;
13 import android.widget.Button;
14 import android.widget.EditText;
15 import android.widget.Toast;
16  publicclass StatusActivity extends Activity implements
   OnClickListener{
17 privatestaticfinal String TAG = "StatusActivity";
18      EditText editText;
19      Button updateButton;
20      Twitter twitter;
21      TextView textCounter;
22      /** Called when the activity is first created. */
23      @Override
24      publicvoid onCreate(Bundle savedInstanceState) {
25      super.onCreate(savedInstanceState);
26      setContentView(R.layout.status);
27      editText= (EditText) findViewById(R.id.editText);
28      updateButton=(Button) findViewById(R.id.buttonUpdate);
29      updateButton.setOnClickListener(this);
30      textCounter= (TextView) findViewById(R.id.textCount);
31      textCounter.setText(Integer.toString(140));
32      textCounter.setTextColor(Color.Green);
33      editText.addTextChangedListener(this);
34      twitter = new Twitter("username", "password");
35      twitter.setAPIRootUrl("http://envy.thinkdigit.com/
   api");}
36 // Button Click
37 publicvoid onClick(View v){
38      String status = editText.getText().toString();
39      new PostToTwitter().execute(status);
40      Log.d(TAG, "onClicked");}
41 //Asynchronous Posting to Twitter
42      class PostToTwitter extends AsyncTask<String, Integer,
   String>{
43 //Call to initiate Background activity
```

```
44      @Override
45      protected String doInBackground(String... statuses){
46      try{
47      Twitter.Status status=twitter.updateStatus(statuses[0]);
48      return status.text;
49      }catch (TwitterException e){
50      Log.e(TAG, e.toString());
51      e.printStackTrace();
52      return "Failed to post";}}
53 //Call when there is a status to update
54      @Override
55      protectedvoid onProgressUpdate(Integer... values){
56      super.onProgressUpdate();}
57 //Call upon completion of Background activity
58      @Override
59      protectedvoid onPostExecute(String result){
60      Toast.makeText(StatusActivity.this, result, Toast.LENGTH_
   LONG).show();}}
61 //Text Watcher methods
62      publicvoid afterTextChanged(Editable statusText){
63      int count = 140 -statusText.length();
64      textCounter.setText(Integer.toString(Count));
65      textCounter.setTextColor(Color.GREEN);
66      if (count<10)
67      textCounter.setTextColor(Color.YELLOW);
68      if (count<0)
69      textCounter.setTextColor(Color.RED);}
70      publicvoid beforeTextChanged(CharSequence s, int start,
   int count, int after){}
71      publicvoid onTextChanged(CharSequence s, int start, int
   before, int count){}}
```

3.9 Adding Color and Graphics

The StatusActivity application looks quite dull. Android offers extensive support for good graphics.

3.9.1 Adding Images

To add a background to the screen, you'll be using some kind of a graphics

file. Most images go to a resource folder called drawable. There'll be three folders with this name in your Package Explorer:

/res/drawable-hdpi for high-density screens

/res/drawable-mdpi for medium-density screens

/res/drawable-ldpi for low-density screens

Create a folder called drawable in the res folder by selecting New > Folder by right-clicking res menu. Add your pictures independent of screen density in this folder with the name backgroung.jpg. Although Android supports many formats, it's advisable to use PNG as it is lossless when compared to GIF.

Now we'll have a reference to R.drawable.background, created by ADT plug-in of Eclipse. We could use this from Java; instead we'll update the status activity layout file res/layout/status.xml. Now, we have two ways of adding the background to the top layout.

3.9.2 Using WYSIWYG Editor

Select the main layout. A highlighted border indicates the selected layout. Open the outline element and select the top element there (or from Window > Show View > Outline). Select the top layout from the Outline view, and you'll notice a border around the entire activity. Next, open the Properties view in Eclipse (Window > Show View > Other), and under the General section, pick Properties. In this view, you can change various properties. To modify the background, click on the "..." button which will open the Reference Chooser dialog. Choose Drawable > Background, which will set the background of the top layout to @drawable/background. Our status.xml layout is referring to the background.jpg drawable.

3.9.3 Updating directly in XML Code

Open the status.xml in the XML editor. To add the background resource to the entire activity, add android:background="@drawable/background" to the <LinearLayout> element.

3.9.4 Adding Color

You can customize the color of backgrounds from the standard RGB color set, or optionally expand it with an Alpha channel. Color can be expressed as RGB or ARGB, where A is the amount of transparency, R is the amount of red, G is for green, and B stands for blue. The combination of these three colors along with the optional transparency gives you the entire spectrum of colors. Each channel can be represented as values between 0 and 255,

or using hex form from #00 to #FF. #3A9F is the same as #33AA99FF and corresponds to #33 for alpha, #AA for red, #99 for green, and #FF for blue. The symbol '#'corresponds to hexadecimal values. You could alternatively write @android:color/<color name> for specifying the color. For instance, white written in place of <color name> generates white color.

3.10 Optimizing the User Interface

The user interface is the primary selling point of any Android application. An app that doesn't look good doesn't sell. To create a simple screen as we have done for the StatusActivity here, the application has to inflate the XML from resources, and create a new Java object for every element and assign its specific properties. Following all this, it needs to draw each widget on the screen. This takes up huge processing time, hence the necessity to optimize the UI. Some optimization points are:

- Limit number of widgets on the screen
- Don't nest unnecessary objects in a loop
- Keep structure as flat as possible, hence it's recommended to use relative layouts.

3.11 Hierarchy Viewer

The Hierarchy Viewer application allows you to attach to any Android device, emulator, or physical phone and reflect over the structure of the current view. It shows all widgets currently loaded in memory, their relationships and properties. This viewer will help you analyse not just one screen, but the screens of any application on your device. This is also a good way to see how some other applications are structured.

4 Accessing location-based services

4.1 Overview

One of the most popular features on current mobile devices is GPS capability. Besides GPS, your phone can also use alternatives like cell tower triangulation and proximity to public Wi-Fi hotspots to identify your location.

While the most prevalent uses of location-based services are for mapping and getting directions, there are other things you can do if you know your location. You can set up a dynamic chat application based on physical location, to chat with people near your location like Buzz on Google Maps. You can also automatically geo-tag posts/pictures to Twitter or similar services.

Android devices may have one or more of these services available to them. You, as a developer, can ask the device for your location, plus details on which providers are available. There are even ways for you to simulate your location in the emulator, for use in testing your location-enabled applications.

4.2 Location providers

Android devices can access several different means of determining your location. The SDK has abstracted all this out into a set of `LocationProvider` objects. Your Android environment will have zero or more `LocationProvider` instances: one for each distinct locating service that is available on the device. Providers know not only your location, but also are aware of their own characteristics, in terms of accuracy, cost, and so on.

You, as a developer, will use a `LocationManager`, which holds the `LocationProvider` set, to figure out which `LocationProvider` is right for your particular circumstance. You'll also need a permission in your application, or the various location APIs will fail due to a security violation. Depending on which location providers you wish to use, you may need `ACCESS_COARSE_LOCATION`, `ACCESS_FINE_LOCATION`, or both.

4.3 Finding yourself

The most obvious thing to do with a location service is to figure out where you are right now. To determine your current location, first you need to get a LocationManager call getSystemService (`LOCATION_SERVICE`) from your activity or service and cast it as a `LocationManager`.

The next step is to get the name of the `LocationProvider` you want to use. Here, you have two main options:

- Ask the user to pick a provider.
- Find the best-match provider based on a set of criteria.

If you want the user to pick a provider, calling `getProviders()` on the `LocationManager` will give you a List of providers, which you can then present to the user for selection. Alternatively, you can create and populate a `Criteria` object, stating the particulars of what you want out of a `LocationProvider`. Here are some of the criteria you can specify:

- `setAltitudeRequired()`: Indicates whether or not you need the current altitude
- `setAccuracy()`: Sets a minimum level of accuracy, in metres, for the position
- `setCostAllowed()`: Controls if the provider must be free or if it can incur a cost on behalf of the device user

Given a filled-in `Critieria` object, call `getBestProvider()` on your LocationManager. Android will sift through the criteria and give you the best answer. Note that not all of your criteria may be met. All, but the monetary cost criterion, might be relaxed if nothing matches.

You're also welcome to hardwire in a `LocationProvider` name (e.g., `GPS_PROVIDER`), perhaps just for testing purposes.

Once you know the name of the `LocationProvider`, you can call `getLastKnownPosition()` to find out where you were recently. Note that "recently" might be fairly out of date (e.g., the phone was turned off) or even null if there has been no location recorded for that provider yet. Calling `getLastKnownPosition()` incurs no monetary or power cost, since the provider doesn't need to be activated to get the value.

This method returns a `Location` object, which can give you the latitude and longitude of the device in degrees as a Java `double`. If the particular location provider offers other data, you can get that as well:

- For altitude, `hasAltitude()` will tell you if there's an altitude value, and `getAltitude()` will return the altitude in metres.
- For bearing (i.e., compass-style direction), `hasBearing()` will tell you if there's a bearing available, and `getBearing()` will return it as degrees to the East of true North.
- For speed, hasSpeed() will tell you if the speed is known and getSpeed() will return the speed in metres per second.

4.4 On the Move

Not all location providers are necessarily immediately responsive. GPS, for example, requires activating a radio and getting a fix from the

satellites before you get a location. That's why Android doesn't offer agetMeMyCurrentLocationNow() method. Combine that with the fact that your users may want their movements to be reflected in your application, and you're probably best off registering for location updates and using that as your means of getting the current location. To register for updates, call requestLocationUpdates() on your LocationManager instance. This method takes four parameters:

- The name of the location provider you wish to use
- How long, in milliseconds, must have elapsed before you might get a location update
- How far, in meters, the device must have moved before you might get a location update
- A LocationListener that will be notified of key location-related events

Here's an example of a LocationListener:

```
LocationListener onLocationChange=new LocationListener() {
public void onLocationChanged(Location location) {
    updateForecast(location);
 }
 public void onProviderDisabled(String provider) {
   // required for interface, not used
 }

 public void onProviderEnabled(String provider) {
   // required for interface, not used
 }

 public void onStatusChanged(String provider, int status,
                             Bundle extras) {
    // required for interface, not used
}
};
```

When you no longer need the updates, call removeUpdates() with the LocationListener you registered. If you fail to do this, your application will continue receiving location updates even after all activities and such are shut down, which will also prevent Android from reclaiming your application's memory.

Sometimes, you're not interested in where you are now, or even when you move, but want to know when you get to where you're going. This could be

an end destination, or it could be getting to the next step on a set of directions. For this, you can give the user the next instruction.

To accomplish this, `LocationManager` offers `addProximityAlert()`. This registers a PendingIntent, which will be fired off when the device gets within a certain distance of a certain location. `TheaddProximityAlert()` method takes the following as parameters:

- The latitude and longitude of the position of interest
- A radius, specifying how close you should be to that position for the Intent to be raised
- A duration for the registration, in milliseconds (after this period, the registration automatically lapses); a value of -1 means the registration lasts until you manually remove it via `removeProximityAlert()`
- The PendingIntent to be raised when the device is within the target zone expressed by the position and radius

Note that there's no guarantee that you'll actually receive an Intent. There may be in an interruption in location services, or the device may not be in the target zone during the period of time the proximity alert is active. For example, if the position is off by a bit, and the radius is a little too tight, the device might only skirt the edge of the target zone, or it may go by the target zone so quickly that the device's location isn't sampled during that time.

It's up to you to arrange for an activity or intent receiver to respond to the Intent you register with the proximity alert. What you do when the Intent arrives is up to you. For example, you might set up a notification (e.g., vibrate the device), log the information to a content provider, or post a message to a web site.

Note that you'll receive the Intent whenever the position is sampled and you're within the target zone, not just upon entering the zone. Hence, you'll get the Intent several times—perhaps quite a few times, depending on the size of the target zone and the speed of the device's movement.

4.5. Testing

The Android emulator doesn't have the ability to get a fix from GPS, triangulate your position from cell towers, or identify your location through nearby Wi-Fi signals. So, if you want to simulate a moving device, you'll need to have some means of providing mock location data to the emulator. One likely option for supplying mock location data is the Dalvik Debug Monitor Service (DDMS). This is an external program, separate from the emulator, which can feed the emulator single location points or full routes to traverse, in a few different formats.

5 Background services

Android conceptualised the Service class to create application components specifically to handle operations and functionality that should run invisibly, without a user interface. It accords Services a higher priority than inactive Activities, so they're less likely to be killed when the system requires resources. In fact, should the run time prematurely terminates a Service that's been started, it can be configured to restart as soon as sufficient resources become available. In extreme cases, the termination of a Service—such as an interruption in music playback—will noticeably affect the user experience, and in these cases a Service's priority can be raised to the equivalent of a foreground activity.

By using Service, you ensure that your applications continue to run and respond to events, even when they're not in active use. They run without a dedicated GUI, but, like Activities and Broadcast Receivers, they still execute in the main Thread of the application's process. To help keep your applications responsive, you'll learn to move time-consuming processes (like network lookups) into background threads using the Thread and AsyncTask classes.

5.1 Services

Unlike Activities, which present a rich graphical interface to users, the Service class runs in the background—updating your Content Providers, firing Intents, and triggering Notifications. They're the perfect means of performing ongoing or regular processing and of handling events even when your application's Activities are invisible or inactive, or have been closed.

Services are started, stopped, and controlled by other application components, including other Services, Activities, and Broadcast Receivers. If your application performs actions that don't depend directly on user input, Services may be the answer.

Since started Services always have higher priority than inactive or invisible Activities, the only reason Android will stop a Service prematurely is to provide additional resources for a foreground component (usually an Activity). When that happens, your Service will automatically restart when resources become available.

If your Service is interacting directly with the user (for example, by playing music) it may be necessary to increase its priority to that of a foreground Activity. This will ensure that your Service isn't terminated

except in extreme circumstances. However, this reduces the run time's ability to manage its resources, potentially degrading the overall user experience.

Applications that update regularly but only rarely or intermittently need user interaction are good candidates for implementation as Services. MP3 players and sports-score monitors are examples of applications that should continue to run and update without a visible Activity.

Further examples can be found within the software stack itself: Android implements several Services, including the Location Manager, Media Controller, and Notification Manager.

5.2 Creating and Controlling Services

In the following sections you'll learn how to create a new Service, and how to start and stop it using Intents and the startService method. Later you'll learn how to bind a Service to an Activity to provide a richer communications interface.

5.2.1 Creating a Service

To define a Service, create a new class that extends Service. You'll need to override onBind and onCreate:

```
import android.app.Service;
    import android.content.Intent;
    import android.os.IBinder;
    public class MyService extends Service {
      @Override
      public void onCreate() {
        // TODO: Actions to perform when service is created.
      }
      @Override
      public IBinder onBind(Intent intent) {
         // TODO: Replace with service binding implementation.
         return null;
      }
  }
```

In most cases, you'll also want to override `onStartCommand`. This is called whenever the Service is started with a call to `startService`, so it may be executed several times within a Service's lifetime. You should ensure that your Service accounts for this.

The `onStartCommand` handler replaces the `onStart` event that was used prior to Android 2.0. By contrast, it enables you to tell the system how to

handle restarts if the Service is killed by the system prior to an explicit call to `stopService` or `stopself`.

The following snippet extends the previous code to show the skeleton code for overriding the `onStartCommand` handler. Note that it returns a value that controls how the system will respond if the Service is restarted after being killed by the run time.

```
@Override
    public int onStartCommand(Intent intent, int flags, int
startId) {
      // TODO Launch a background thread to do processing.
      return Service.START_STICKY;
    }
```

Services are launched on the main Application thread, meaning that any processing done in the `onStartCommand` handler will happen on the main GUI thread. The standard pattern for implementing a Service is to create and run a new Thread from `onStartCommand` to perform the processing in the background and stop the Service when it's complete.

This pattern lets `onStartCommand` complete quickly, and lets you control the restart behavior using one of the following Service constants:

- `START_STICKY` describes the standard behaviour, which is similar to the way in which `onStart` was implemented prior to Android 2.0. If you return this value, `onStartCommand` will be called any time your Service restarts after being terminated by the run time. Note that on restart the intent parameter passed on to `onStartCommand` will be null.
- This mode is typically used for Services that handle their own states, and that are explicitly started and stopped as required (via `startService` and `stopService`). This includes Services that play music or handle other ongoing background tasks.
- `START_NOT_STICKY` is used for Services that are started to process specific actions or commands. Typically they'll use `stopself` to terminate once that command has been completed.
- Following termination by run time, Services set to this mode will restart only if there are pending start calls. If no `startService` calls have been made since the Service was terminated, the Service will be stopped without a call being made to `onStartCommand`.
- This mode is ideal for Services that handle specific requests, particularly regular processing such as updates or network polling. Rather than restarting the Service during a period of resource

contention, it's often more prudent to let the Service stop and retry at the next scheduled interval.

- `START_REDELIVER_INTENT` is used in circumstances you'll want to ensure that the commands you have requested from your Service are completed.
- This mode is a combination of the first two—if the Service is terminated by run time, it will restart only if there are pending start calls or the process was killed prior to its calling `stopself`.
- In the latter case, a call to `onStartCommand` will be made, passing the initial Intent whose processing did not properly complete.

The restart mode you specify in your `onStartCommand` return value will affect the parameter values passed to subsequent calls. Initially the Intent will be the parameter you passed to `startService` to start your Service. After system-based restarts it will be either null, in the case of `START_STICKY` mode, or the original Intent, if the mode is set to `START_REDELIVER_INTENT`. Use the flag parameter to discover how the Service was started. In particular, you can use the code snippet below to determine if either of the following cases is true:

- `START_FLAG_REDELIVERY` indicates that the Intent parameter is a redelivery caused by the system run time's having terminated the Service before it was explicitly stopped by a call to stopSelf.
- `START_FLAG_RETRY` indicates that the Service has been restarted after an abnormal termination. Passed when the Service was previously set to `START_STICKY`.

```
@Override
      public int onStartCommand(Intent intent, int flags, int
startId) {
       if ((flags &START_FLAG_RETRY) == 0) {
          // TODO If it's a restart, do something.
       }
       else {
         // TODO Alternative background process.
       }
       return Service.START_STICKY;
    }
```

5.2.2 Registering a Service in the Manifest

Once you've constructed a new Service, you must register it in the application manifest. Do this by including a <service> tag within the application node.

Use the requires-permission attribute to require a uses-permission for other applications to access this Service.
The following is the service tag you'd add for the skeleton Service you created earlier:

```
<service android:enabled="true" android:name=".MyService"/>
```

5.2.3 Self-Terminating a Service

Once your Service has completed the actions or processing it was started, make a call to stopSelf, either without a parameter to force a stop, or by passing a startId value to ensure processing has been completed for each instance of startService called so far, as shown in the following snippet:

```
stopSelf (startId);
```

By explicitly stopping the Service when your processing is complete, you allow the system to recover the resources otherwise required to keep it running. Due to the high priority of Services they're not commonly killed by run time, so self-termination can significantly improve the resource footprint of your application.

5.2.4 Starting, Controlling and Interacting with a Service

To start a Service, call startService; you can either use an action to implicitly start a Service with the appropriate Intent Receiver registered, or you can explicitly specify the Service using its class. If the Service requires permissions that your application doesn't have, the call to startService will throw a SecurityException.

In both cases, you can pass values to the Service's onStart handler by adding extras to the Intent, which demonstrates both techniques available for starting a Service:

```
// Implicitly start a Service
    Intent myIntent = new Intent(MyService.ORDER_PIZZA);
    myIntent.putExtra("TOPPING", "Margherita");
    startService(myIntent);
    // Explicitly start a Service
    startService(new Intent(this, MyService.class));
```

To stop a Service use stopService, passing an Intent that defines the Service to stop. The following code snippet first starts and then stops a Service both, explicitly and by using the component name returned from a call tostartService.

```
ComponentName service = startService(new Intent(this,
```

```
BaseballWatch.class));
       // Stop a service using the service name.
       stopService(new Intent(this, service.getClass()));
       // Stop a service explicitly.
       try {
       Class serviceClass = Class.forName(service.getClassName());
         stopService(new Intent(this, serviceClass));
       } catch (ClassNotFoundException e) {}
```

If startService is called on a Service that's already running, the Service's onStartCommand handler will be executed again. Calls to startService do not nest, so a single call to stopService will terminate it no matter how many times startService has been called.

5.3 Binding Activities to Services

When an Activity is bound to a Service, it maintains a reference to the Service instance itself, enabling you to make method calls on the running Service as you would on any other instantiated class.

Binding is available for Activities that would benefit from a more detailed interface with a Service. To support binding for a Service, implement the onBind method:

```
private final IBinder binder = new MyBinder();

    @Override
    public IBinder onBind(Intent intent) {
      return binder;
    }

    public class MyBinder extends Binder {
      MyService getService() {
        return MyService.this;
      }
  }
```

The connection between the Service and Activity is represented as a ServiceConnection. You'll need to implement a new ServiceConnection, overriding the onServiceConnected and onServiceDisconnected methods to get a reference to the Service instance once a connection has been established:

```
// Reference to the service
    private MyService serviceBinder;
```

```
    // Handles the connection between the service and activity
  private ServiceConnection mConnection = new ServiceConnection()
{
       public void onServiceConnected(ComponentName className,
IBinder service) {
        // Called when the connection is made.
              serviceBinder = ((MyService.MyBinder)service).
getService();
      }

     public void onServiceDisconnected(ComponentName className)
{
        // Received when the service unexpectedly disconnects.
        serviceBinder = null;
      }
   };
```

To perform the binding, call bindService, passing an Intent (either explicit or implicit) that selects the Service to bind to an instance of your new ServiceConnection implementation:

```
  @Override
    public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      // Bind to the service
 Intent bindIntent = new Intent(MyActivity.this, MyService.
class);
       bindService(bindIntent, mConnection, Context.BIND_AUTO_
CREATE);
   }
```

Once the Service has been bound, all of its public methods and properties are available through the serviceBinder object obtained from the onServiceConnected handler.

Android applications don't (normally) share memory, but in some cases your application may want to interact with (and bind to) Services running in different application processes.

You can communicate with a Service running in a different process using broadcast Intents or through the extras Bundle in the Intent used to start the Service. If you need a more tightly coupled connection you can make a Service available for binding across application boundaries using Android Interface Definition Language (AIDL). AIDL defines the Service's interface in terms of OS level primitives, allowing Android to transmit objects across process boundaries.

5.4 Prioritising Background Services

When calculating which applications and application components should be killed, Android assigns running Services the second-highest priority. Only active, foreground Activities are considered a higher priority in terms of system resources. In extreme cases, in which your Service is interacting directly with the user, it may be appropriate to lift its priority to the equivalent of a foreground Activity's. Do this by setting your Service to run in the foreground using the startForeground method.

Services running in the foreground may interactg directly with the user (for example, by playing music). Because of this, the user should always be aware of a foreground Service. To ensure this, calls to startForeground must specify an ongoing Notification. This Notification will continue for at least as long as the Service is running in the foreground.

```
int NOTIFICATION_ID = 1;

    Intent intent = new Intent(this, MyActivity.class);
      PendingIntent pi = PendingIntent.getActivity(this, 1,
intent, 0));
     Notification notification = new Notification(R.drawable.
icon,
      "Running in the Foreground", System.currentTimeMillis());
    notification.setLatestEventInfo(this, "Title", "Text", pi);

  notification.flags = notification.flags |
                       Notification.FLAG_ONGOING_EVENT;
    startForeground(NOTIFICATION_ID, notification);
```

This code uses setLatestEventInfo to update the Notification using the default status window layout. Later in this chapter, you'll learn how to specify a custom layout for your Notification. Using this technique, you can provide more details of your ongoing Service to users.

Once your Service no longer requires foreground priority you can move it back to the background, and optionally remove the ongoing Notification using the stopForeground method. The Notification will be automatically cancelled if your Service stops or is terminated.

```
// Move to the background and remove the Notification
    stopForeground(true);
```

5.5 Using Background Threads

To ensure that your applications remain responsive, it's good practice to move all slow, time-consuming operations off the main application thread and onto a child thread.

Android provides two alternatives for backgrounding your processing. The AsyncTask class lets you define an operation to be performed in the background, then provides event handlers you can use to monitor progress and post the results on the GUI thread. Alternatively, you can implement your own Threads and use the Handler class to synchronize with the GUI thread before updating the UI. Both techniques are described in this section.

5.5.1 Using AsyncTask to Run Asynchronous Tasks

The AsyncTask class offers a simple, convenient mechanism for moving your time-consuming operations onto a background thread. It offers the convenience of event handlers synchronized with the GUI thread to let you update Views and other UI elements to report progress or publish results when your task is complete.

AsyncTask handles all of the Thread creation, management and synchronization, enabling you to create an asynchronous task consisting of processing to be done in the background and a UI update to be performed when processing is complete.

Creating a new Asynchronous Task

To create a new asynchronous task, you need to extend AsyncTask. Your implementation should specify the classes used for input parameters on the execute method, the progress-reporting values and the result values in the following format:

```
    AsyncTask<[Input Parameter Type], [Progress Report Type],
[Result Type]>
```

If you don't need or want to take input parameters, update progress, or report a final result, simply specify Void for any or all of the types required.

```
  private class MyAsyncTask extends AsyncTask<String, Integer,
Integer> {
        @Override
        protected void onProgressUpdate(Integer... progress) {
          // [... Update progress bar, Notification, or other UI
element ...]
        }

        @Override
        protected void onPostExecute(Integer... result) {
             // [... Report results via UI update, Dialog, or
notification ...]
        }
        @Override
        protected Integer doInBackground(String... parameter) {
          int myProgress = 0;
           // [... Perform background processing task, update
myProgress ...]
        PublishProgress(myProgress)
       // [... Continue performing background processing task ...]

        // Return the value to be passed to onPostExecute
        return result;
      }
  }
```

As shown in the code, your subclass should implement the following event handlers:

- doinBackground Takes a set of parameters of the type defined in your class implementation. This method will be executed on the background thread, so it must not attempt to interact with UI objects.
- Place your long-running code here, using the publishProgress method to allow onProgressUpdate to post progress updates to the UI.
- When your background task is complete, return the final result for the onPostExecute handler to report it to the UI.
- onProgressUpdate Override this handler to post interim updates to the UI thread. This handler receives the set of parameters passed to publishProgress from within doinBackground.

- This handler is synchronized with the GUI thread when executed, so you can safely modify UI elements.
- onPostExecute When doinBackground has completed, the return value from that method is passed to this event handler.
- Use this handler to update the UI once your asynchronous task has completed. onPostExecute is synchronized with the GUI thread when executed, so you can safely modify UI elements.

Running an Asynchronous Task

Once you've implemented your asynchronous task, execute it by creating a new instance and by calling execute. You can pass in a number of parameters, each of the type specified in your implementation.

new MyAsyncTask().execute("inputString1", "inputString2");

5.5.2 Manual Thread Creation and GUI Thread Synchronization

While AsyncTask is a useful shortcut, there are times when you'll want to create and manage your own Threads to perform background processing. In this section you'll learn how to create and start new Thread objects, and how to synchronize with the GUI thread before updating the UI.

Creating a New Thread

You can create and manage child threads using Android's Handler class and the threading classes available within java.lang.Thread. The following code is for moving processing onto a child thread.

```
// This method is called on the main GUI thread.
   private void mainProcessing() {
         // This moves the time consuming operation to a child
thread.
Thread thread = new Thread(null, doBackgroundThreadProcessing,
                                        "Background");
        thread.start();
   }

   // Runnable that executes the background processing method.
private Runnable doBackgroundThreadProcessing = new Runnable() {
        public void run() {
          backgroundThreadProcessing();
        }
   };
```

```
    // Method which does some processing in the background.
    private void backgroundThreadProcessing() {
        [ ... Time consuming operations ... ]
    }
```

Using the Handler for Performing GUI Operations

Whenever you're using background threads in a GUI environment it's important to synchronize child threads with the main application (GUI) thread before creating or modifying graphical elements.

Within your application components, Notifications and Intents are always received and handled on the GUI thread. In all other cases, operations that explicitly interact with objects created on the GUI thread (such as Views) or that display messages (like Toasts) must be invoked on the main thread.

If you're running within an Activity, you can also use the runOnUiThread method, which lets you force a method to execute on the same Thread as the Activity UI.

```
runOnUiThread(new Runnable() {
       public void run() {
         // TODO Update a View.
       }
    });
```

In other circumstances (such as Toasts and Notifications) you can use the Handler class to post methods onto the Thread in which the Handler was created.

Using the Handler class you can post updates to the user interface from a background thread using the Post method. The following code shows the outline for using the Handler to update the GUI thread.

```
// Initialize a handler on the main thread.
    private Handler handler = new Handler();

    private void mainProcessing() {
     Thread thread = new Thread(null, doBackgroundThreadProcessing,
                                      "Background");
      thread.start();
    }

   private Runnable doBackgroundThreadProcessing = new Runnable()
{
        public void run() {
```

```
            backgroundThreadProcessing();
        }
    };

    // Method which does some processing in the background.
    private void backgroundThreadProcessing() {
        [ ... Time consuming operations ... ]
handler.post(doUpdateGUI);
    }

// Runnable that executes the update GUI method.
    private Runnable doUpdateGUI = new Runnable() {
        public void run() {
        updateGUI();
        }
    };

    private void updateGUI() {
      [ ... Open a dialog or modify a GUI element ... ]
    }
```

The Handler class also lets you delay posts or execute them at a specific time, using the postDelayed and postAtTime methods, respectively.

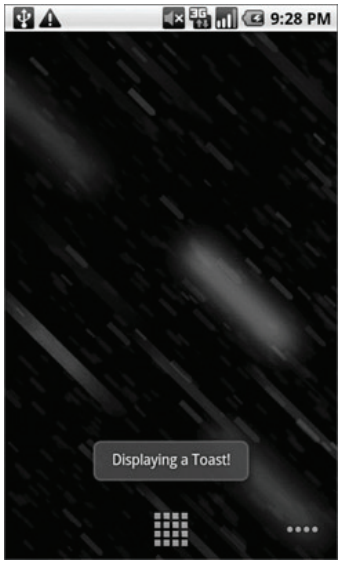
5.6 Toast

Toasts are transient dialog boxes that remain visible for only a few seconds before fading out. Toasts don't steal focus and are non-modal, so they don't interrupt the active application. These dialog boxes are perfect for informing your users of events without forcing them to open an Activity or read a Notification. They provide an ideal mechanism for alerting users to events occurring in background Services without interrupting foreground applications.

The Toast class includes a static makeText method that creates a standard Toast display window. Pass the application Context, the text message to display and the length of time to display it (LENGTH_SHORT orLENGTH_LONG) to the makeText method to construct a new Toast. Once a Toast has been created, display it by calling show.

```
Context context = getApplicationContext();
    String msg = "To health and happiness!";
```

```
   int duration = Toast.LENGTH_SHORT;
Toast toast = Toast.makeText(context,
msg, duration);
toast.show();
```



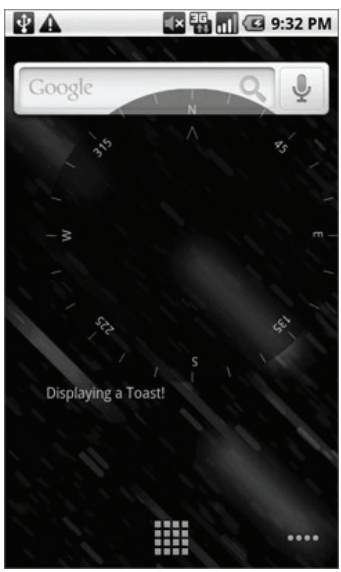
It will remain on screen for around two seconds before fading out. The application behind it remains fully responsive and interactive while the Toast is visible.

5.6.1 Customizing Toasts

The standard Toast text message window is often sufficient, but in many situations you'll want to customize its appearance and screen position. You can modify a Toast by setting its display position and assigning it alternative Views or layouts.

The following code shows how to align a Toast to the bottom of the screen using the setGravity method.

```
Context context = getApplicationContext();
    String msg = "To the bride and
groom!";
   int duration = Toast.LENGTH_SHORT;
 Toast toast = Toast.makeText(context,
msg, duration);
    int offsetX = 0;
    int offsetY = 0;

toast.setGravity(Gravity.BOTTOM,
offsetX, offsetY);
    toast.show();
```



When a text message just isn't going to get the job done, you can specify a custom View or layout to use a more complex, or more visual, display. Using setView on a Toast object, you can specify any View (including a layout) to display using the transient message window mechanism.

5.6.2 Using Toasts in Worker Threads

As GUI components, Toasts must be opened on the GUI thread or you risk throwing a cross-thread exception. In the following code, a Handler is used to ensure that the Toast is opened on the GUI thread.

```
 private void mainProcessing() {
    Thread thread = new Thread(null, doBackgroundThreadProcessing,
                                        "Background");
        thread.start();
    }

   private Runnable doBackgroundThreadProcessing = new Runnable()
{
        public void run() {
          backgroundThreadProcessing();
        }
    };

   private void backgroundThreadProcessing() {
handler.post(doUpdateGUI);
    }

    // Runnable that executes the update GUI method.
    private Runnable doUpdateGUI = new Runnable() {
        public void run() {
Context context = getApplicationContext();
          String msg = "To open mobile development!";
          int duration = Toast.LENGTH_SHORT;
          Toast.makeText(context, msg, duration).show();
        }
    };
```

6 Alerting Users via Notifications

6.1 Overview

Android has a whole framework for dealing with pop-up messages, flashing lights, vibration alerts and sound alerts, collectively called notifications, which are handled by the Notification Manager. All Broadcast Receivers, Services and inactive Activities that alert users of events that require attention—for example, incoming calls and appointment reminders—are part of this. A Notification is the only way an application can grab the user's attention if the app is open in the background, and the user isn't paying attention to it.

Applications can add their own status bar icons, with care taken to have them appear only when needed. This can be persisted through insistent repetition. Status bar icons can also be updated regularly or expanded to show additional information using the expanded status bar window.

To raise notifications via the NotificationManager, you need to get the service object via getSystemService(NOTIFICATION_SERVICE) from your activity. The NotificationManager gives you three methods: one to pester (notify()) and two to stop pestering (cancel() and cancelAll()). Using the Notification Manager, you can trigger new notifications, modify existing ones or remove those that are no longer needed or wanted.

6.1.1 Hardware Notifications

You can flash LEDs on the device by setting lights to true, also specifying the color (as an #ARGB value in ledARGB) and the pattern in which the light should blink (by providing off/on durations in milliseconds for the light via ledOnMS and ledOffMS).

The following code snippet shows how to turn on the red LED device:

```
notification.ledARGB = Color.RED;
notification.ledOffMS = 0;
notification.ledOnMS = 1;
notification.flags = notification.flags | Notification.FLAG_
SHOW_LIGHTS;
```

Android lets you play any audio file on the phone as a Notification by assigning a location URI to the sound property, as shown in the snippet below:

```
notification.sound = ringURI;
```

You can also vibrate the device, controlled via a long[] indicating the on/off patterns (in milliseconds) for the vibration (vibrate). Even the pattern of a vibration is under your control. To set a vibration pattern, assign an array of longs to the Notification's vibrate property. Construct the array so that every alternate number is the length of time (in milliseconds) to vibrate or pause, respectively.

To use vibration in your application, you need to be granted permission. Add a uses-permission to your application to request access to the device vibration using the following code snippet:

```
<uses-permissionandroid:name="android.permission.VIBRATE"/>
```

The following example shows how to modify a Notification to vibrate in a repeating pattern of 1 second on, 1 second off, for 5 seconds in total:

```
long[] vibrate = new long[] { 1000, 1000, 1000, 1000, 1000 };
notification.vibrate = vibrate;
```

6.1.2 Icons

To set up an icon for a Notification, create a new Notification object, passing in the icon to display in the status bar, along with the status bar ticker-text and the time of this Notification:

```
int icon = R.drawable.icon;
String tickerText = "Notification";
long when = System.currentTimeMillis();
Notificationnotification = new Notification(icon, tickerText,
when);
```

Use the number property to display the number of events a status bar icon represents.

```
notification.number++;
```

6.2 Triggering Notifications

To fire a Notification, pass it in to the notify method on the NotificationManager along with an integer reference ID:

```
intnotificationRef = 1;
notificationManager.notify(notificationRef, notification);
```

To cancel a Notification-

```
notificationManager.cancel(notificationRef);
```

Cancelling a Notification removes its status bar icon and clears it from the extended status window.

6.3 Ongoing and Insistent Notifications

Notifications can be configured as ongoing and/or insistent by setting the `FLAG_INSISTENT` and `FLAG_ONGOING_EVENT` flags.
notification.flags = notification.flags
Notification.FLAG_ONGOING_EVENT;
Insistent Notifications repeat continuously until cancelled.
notification.flags = notification.flags
Notification.FLAG_INSISTENT;
Insistent Notifications are handled by continuously repeating the initial Notification effects until the Notification is cancelled.

6.4 SeeingNotifications in Action

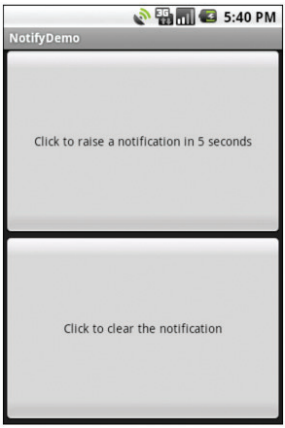
Let's now take a peek at a sample NotifyDemo class:
packagecom.commonsware.android.notify;

```
importandroid.app.Activity;
importandroid.app.Notification;

importandroid.app.NotificationManager;
importandroid.app.PendingIntent;
importandroid.content.Intent;
importandroid.os.Bundle;
importandroid.view.View;
importandroid.widget.Button;
importjava.util.Timer;
importjava.util.TimerTask;

public class NotifyDemo extends Activity {
private static final int NOTIFY_ME_ID=1337;
private Timer timer=new Timer();
privateint count=0;

  @Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);

  Button btn=(Button)findViewById(R.id.notify);
```

```
btn.setOnClickListener(new View.OnClickListener() {
public void onClick(View view) {
TimerTask task=new TimerTask() {
public void run() {
notifyMe();
           }
         };

timer.schedule(task, 5000);
       }
     });

btn=(Button)findViewById(R.id.cancel);

btn.setOnClickListener(new View.OnClickListener() {
public void onClick(View view) {
NotificationManagermgr=
              (NotificationManager)getSystemService(NOTIFICATION_
SERVICE);

mgr.cancel(NOTIFY_ME_ID);
       }
     });
  }

private void notifyMe() {
finalNotificationManagermgr=
     (NotificationManager)getSystemService(NOTIFICATION_SERVICE);
     Notification note=new Notification(R.drawable.red_ball,
                                              "Status message!",
System.currentTimeMillis());
PendingIntent i=PendingIntent.getActivity(this, 0,
new Intent(this, NotifyMessage.class),
                                                   0);

note.setLatestEventInfo(this, "Notification Title",
       "This is the notification message", i);
note.number=++count;
```

```
mgr.notify(NOTIFY_ME_ID, note);
   }
}
```

This activity sports two large buttons: one to kick off a Notification after a 5-second delay and one to cancel that notification (if it is active).

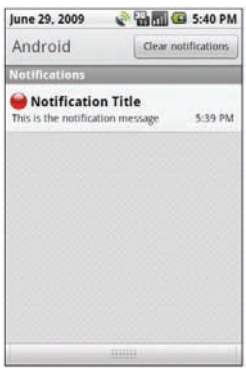


The NotifyDemo activity main view

Creating the Notification, in `notifyMe()`, is accomplished in six steps:

1. Get access to the Notification Manager instance.
2. Create a Notification object, a message to flash on the status bar as the Notification is raised and the time associated with this event.
3. Create a PendingIntent that will trigger the display of another activity (NotifyMessage).
4. Use setLatestEventInfo() to specify that when the notification is clicked we are to display a certain title and message, and if that's clicked, we launch the PendingIntent.
5. Update the number associated with the Notification.
6. Tell the NotificationManager to display the Notification.

Our notification as it appears on the status bar, with our status message



Notification messages

The notifications drawer, fully expanded, with our notification.

7 Telephony and SMS

Let's see how to use Android's telephony APIs to monitor mobile voice and data connections as well as incoming and outgoing calls, and to send and receive SMS messages.

Android provides full access to SMS functionality, letting you send and receive SMS messages from within your applications. Using the Android APIs, you can create your own SMS client application to replace the native clients available as part of the software stack. Alternatively, you can incorporate the messaging functionality within your own applications to create social applications using SMS as the transport layer.

7.1 Handling Telephone Calls

Android has the means to let you manipulate the phone just like any other piece of the Android system.

7.2 Report to the Manager

To get at much of the phone API, you use the TelephonyManager. That class lets you do things like the following:

- Determine if the phone is in use via getCallState(), with return values of CALL_STATE_IDLE (phone not in use), CALL_STATE_RINGING (call requested but still being connected), and CALL_STATE_OFFHOOK (call in progress).
- Find out the SIM ID (IMSI) via getSubscriberId().
- Find out the phone type (e.g., GSM) via getPhoneType(), or find out the data connection type (e.g., GPRS or EDGE) via getNetworkType().

You can also initiate a call from your application, such as from a phone number you obtained through your own web service. To do this, simply craft an ACTION_DIAL Intent with a Uri of the form tel:NNNNN (where NNNNN is the phone number to dial) and use that Intent with startActivity(). This will not actually dial the phone; rather, it activates the dialer activity, from which the user can press a button to place the call.

For example, let's look at the Phone/Dialer sample application.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
android:orientation="vertical"
```

```
android:layout_width="fill_parent"
android:layout_height="fill_parent"
>
<LinearLayout
android:orientation="horizontal"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
>
<TextView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Number to dial:"
      />
<EditText android:id="@+id/number"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:cursorVisible="true"
android:editable="true"
android:singleLine="true"
    />
</LinearLayout>
<Button android:id="@+id/dial"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:layout_weight="1"
android:text="Dial It!"
  />
</LinearLayout>
```

We have a labeled field for typing in a phone number, plus a button for dialing that number. The Java code simply launches the dialer using the phone number from the field:

```
package com.commonsware.android.dialer;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
```

```
import android.view.View;
import android.widget.Button;
import android.widget.EditText;

public class DialerDemo extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);

    final EditText number=(EditText)findViewById(R.id.number);
    Button dial=(Button)findViewById(R.id.dial);

    dial.setOnClickListener(new Button.OnClickListener() {
      public void onClick(View v) {
                                    String
toDial="tel:"+number.
getText().toString();

        startActivity(new
Intent(Intent.ACTION_
DIAL,
                                  Uri.
parse(toDial)));
      }
    });
  }
}
```

The DialerDemo sample application, as initially launched

The Android Dialer activity, as launched from DialerDemo

7.3 SMS

SMS technology is designed to send short text messages between mobile phones. It provides support for sending both, text messages (designed to be read by people) and data messages (meant to be consumed by applications). More recently, MMS (multimedia messaging service) messages have allowed users to send and receive messages containing multimedia attachments such as photos, videos and audio.

Android provides full SMS functionality from within your applications through the SMSManager. Using the SMS Manager, you can replace the

native SMS application to send text messages, react to incoming texts or use SMS as a data transport layer.

Let's demonstrate how to use both, the SMS Manager and Intents to send messages from within your applications.

7.3.1 Sending SMS and MMS from your Application using Intents and the Native Client

In many circumstances, you may find it easier to pass on the responsibility for sending SMS and MMS messages to another application, rather than implementing a full SMS client within your app.

To do so, call startActivity using an Intent with the Intent.ACTION_SENDTO action. Specify a target number using sms: schema notation as the Intent data. Include the message you want to send within the Intent payload using an sms_body extra.

```
    Intent smsIntent = new Intent(Intent.ACTION_SENDTO,
                                  Uri.parse("sms:55512345"));
smsIntent.putExtra("sms_body", "Press send to send me");
     startActivity(smsIntent);
```

You can also attach files (effectively creating an MMS message) to your messages. Add an Intent.EXTRA_STREAM with the URI of the resource to attach, and set the Intent type to the mime-type of the attached resource. Note that the native MMS application doesn't include an Intent Receiver for ACTION_SENDTO with a type set. Instead, you'll need to use ACTION_SEND and include the target phone number as an address extra.

```
// Get the URI of a piece of media to attach.
    Uri attached_Uri = Uri.parse("content ://media/external/
images/media/1");
   // Create a new MMS intent
    Intent mmsIntent = new Intent(Intent.ACTI0N_SEND, attached_
Uri);
     mmsIntent.putExtra("sms_body", "Please see the attached
image");
   mmsIntent.putExtra("address", "07912355432");
   mmsIntent.putExtra(Intent.EXTRA_STREAM, attached_Uri);
   mmsIntent.setType("image/png");
   startActivity(mmsIntent);
```

7.3.2 Sending SMS Messages Manually

SMS messaging in Android is handled by the SmsManager. To send SMS

messages, your applications must specify the SEND_SMS uses-permission. To request this permission, add it to the manifest as shown below:

```
<uses-permission android:name="android.permission.SEND_SMS"/>
```

To send a text message, use sendTextMessage from the SMS Manager, passing in the address (phone number) of your recipient and the text message you want to send.

```
String sendTo = "5551234";
    String myMessage = "Android supports programmatic SMS messaging!";
SmsManager.sendTextMessage(sendTo, null, myMessage, null, null);
```

The second parameter can be used to specify the SMS service center to use; if you enter null, the default service center will be used for your carrier.

The final two parameters let you specify Intents to track the transmission and successful delivery of your messages.

7.3.3 Tracking and Confirming SMS Message Delivery

To track the transmission and delivery success of your outgoing SMS messages, implement and register Broadcast Receivers that listen for the actions you specify when creating the Pending Intents you pass in to thesendTextMessage method.

The first PendingIntent parameter, sentIntent, is fired when the message either is successfully sent or fails to send. The result code for the Broadcast Receiver that receives this Intent will be one of the following:

- **Activity.RESULT_OK:** To indicate a successful transmission
- **SmsManager.RESULT_ERROR_GENERIC_FAILURE:** To indicate a nonspecific failure
- **SmsManager.RESULT_ERROR_RADIO_OFF:** When the phone radio is turned off
- **SmsManager.RESULT_ERROR_NULL_PDU:** To indicate a PDU (protocol description unit) failure

The second PendingIntent parameter, deliveryIntent, is fired only after the destination recipient receives your SMS message.

A typical pattern for sending an SMS and monitoring the success of its transmission and delivery looks like this:

```
String SENT_SMS_ACTION = "SENT_SMS_ACTION";
    String DELIVERED_SMS_ACTION = "DELIVERED_SMS_ACTION";


    // Create the sentintent parameter
```

```
        Intent sentintent = new Intent(SENT_SMS_ACTION);
        PendingIntent sentPI = PendingIntent.getBroadcast(getApp
licationContext(),
                                                          sentIntent,
                                                                  0);
        // Create the deliveryIntent parameter
        Intent deliveryIntent = new Intent(DELIVERED_SMS_ACTION);
        PendingIntent deliverPI =
          PendingIntent.getBroadcast(getApplicationContext(),
                                     0,
                                     deliveryIntent,
                                     0);

        // Register the Broadcast Receivers
         registerReceiver(new BroadcastReceiver() {
                             @Override
                          public void onReceive(Context _context,
Intent _intent)
                             {
                               switch (getResultCode()) {
                                 case Activity.RESULT_OK:
                                     [ ... send success actions
... ] ; break;
                                  case SmsManager.RESULT_ERROR_
GENERIC_FAILURE:
                                  [ ... generic failure actions
... ] ; break;
                                  case SmsManager.RESULT_ERROR_
RADIO_OFF:
                                [ ... radio off failure actions
... ]; break;
                                  case SmsManager.RESULT_ERROR_
NULL_PDU:
                                 [ ... null PDU failure actions
... ] ; break;
                               }
                             },
```

```
                        new IntentFilter(SENT_SMS_ACTION));

        registerReceiver(new BroadcastReceiver() {
                            @Override
                        public void onReceive(Context _context,
Intent _intent)
                            {
                              [ ... SMS delivered actions ... ]
                            }
                          },
                       new IntentFilter(DELIVERED_SMS_ACTION));

        // Send the message
           smsManager.sendTextMessage(sendTo, null, myMessage,
sentPI, deliverPI);
```

7.3.4 Conforming to the Maximum SMS Message Size

SMS text messages are normally limited to 160 characters, so longer messages need to be broken into a series of smaller parts. The SMS Manager includes the divideMessage method, which accepts a string as an input and breaks it into an ArrayList of messages, wherein each is less than the maximum allowable size.

You can then use the sendMultipartTextMessage method on the SMS Manager to transmit the array of messages:

```
  ArrayList<String> messageArray = smsManager.
divideMessage(myMessage);
                ArrayList<PendingIntent> sentIntents = new
ArrayList<PendingIntent>();
          for (int i = 0; i <messageArray.size(); i + +)
             sentIntents.add(sentPI);

          smsManager.sendMultipartTextMessage(sendTo,
                                              null,
                                              messageArray,
                                          sentIntents, null);
```

The sentIntent and deliveryIntent parameters in the sendMultipartTextMessage method are ArrayLists that can be used to specify different Pending Intents to fire for each message part.

7.3.5 Sending Data Messages

You can send binary data via SMS using the sendDataMessage method on an SMS Manager. The sendDataMessage method works much like sendTextMessage, but includes additional parameters for the destination port and an array of bytes that constitutes the data you want to send. The basic structure of sending a data message looks like this:

```
Intent sentIntent = new Intent(SENT_SMS_ACTION);
      PendingIntent sentPI = PendingIntent.getBroadcast(getApp
licationContext(),
                                                          0,
sentIntent, 0);

      short destinationPort = 80;
      byte[] data = [ ...  your data ... ];
      smsManager.sendDataMessage(sendTo, null, destinationPort,
                                 data, sentPI, null);
```

7.3.6 Listening for Incoming SMS Messages

When a new SMS message is received by the device, a new Broadcast Intent is fired with the android.provider.Telephony.SMS_RECEIVED action. Note that this is a string literal. The SDK currently doesn't include a reference to this string, so you must specify it explicitly when using it in your applications.

For an application to listen for SMS Broadcast Intent, it needs to specify the RECEIVE_SMS manifest permission. Request this permission by adding a <uses-permission> tag to the application manifest, as shown in the following snippet:

```
<uses-permission
android:name="android.permission.RECEIVE_SMS"
      />
```

The SMS Broadcast Intent includes the incoming SMS details. To extract the array of SmsMessage objects packaged within the SMS Broadcast Intent bundle, use the PDU extras key to extract an array of SMS PDUs (protocol description units—used to encapsulate an SMS message and its metadata), each of which represents an SMS message. To convert each PDU byte array into an SMS Message object, call SmsMessage.createFromPdu, passing in each byte array:

```
Bundle bundle = intent.getExtras();
     if (bundle != null) {
```

```
        Object[] pdus = (Object[]) bundle.get("pdus");
        SmsMessage[] messages = new SmsMessage[pdus.length];
        for (int i = 0; i < pdus.length; i++)
              messages[i] = SmsMessage.createFromPdu((byte[])
pdus[i]);
      }
```

Each SmsMessage contains the SMS message details, including the originating address (phone number), time stamp and the message body.

Let's see a Broadcast Receiver implementation whose onReceive handler checks incoming SMS texts that start with the string @echo, and then sends the same text back to the number that sent it.

```
      public class IncomingSMSReceiver extends BroadcastReceiver
{
        private static final String queryString = "@echo";
        private static final String SMS_RECEIVED =
          "android.provider.Telephony.SMS_RECEIVED";

       public void onReceive(Context _context, Intent _intent) {
  if (_intent.getAction().equals(SMS_RECEIVED)) {
             SmsManager sms = SmsManager.getDefault();

  Bundle bundle = _intent.getExtras();
             if (bundle != null) {
               Object[] pdus = (Object[]) bundle.get("pdus");
                  SmsMessage[] messages = new SmsMessage[pdus.
length];
                for (int i = 0; i < pdus.length; i++)
                 messages[i] = SmsMessage.createFromPdu((byte[])
pdus[i]);

                for (SmsMessage message : messages) {
  String msg = message.getMessageBody();
                   String to = message.getOriginatingAddress();

                if (msg.toLowerCase().startsWith(queryString)) {
                        String out = msg.substring(queryString.
length());
                 sms.sendTextMessage(to, null, out, null, null);
```

```
                    }
                }
            }
        }
    }
}
```

To listen for incoming messages, register the Broadcast Receiver using an Intent Filter that listens for the android.provider.Telephony.SMS_RECEIVED action String:

```
final String SMS_RECEIVED = "android.provider.Telephony.SMS_
RECEIVED";
        IntentFilter filter = new IntentFilter(SMS_RECEIVED);
        BroadcastReceiver receiver = new IncomingSMSReceiver();
        registerReceiver(receiver, filter);
```

7.4 Emergency Responder SMS Example

In this example, you'll create an SMS application that turns an Android phone into an emergency response beacon. Once complete, the next time you're caught in an unfortunate situation like the 26/11 Mumbai Terror Attacks or the Railway Motormen strike, you can set your phone to automatically respond to your friends' and family members' pleas for a status update with a friendly message (or a desperate cry for help).

To make the task of finding you easier, you'll use location-based services to tell your friends and family exactly where to find you. The robustness of SMS network infrastructure makes SMS an excellent option for applications like this for which reliability and accessibility are critical.

1. Start by creating a new EmergencyResponder project that features an EmergencyResponder Activity.

```
package com.paad.emergencyresponder;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Locale;
import java.util.concurrent.locks.ReentrantLock;
import java.util.List ;
import android.app.Activity;
import android.app.PendingIntent;
import android.content.Context;
```

```
import android.content.Intent ;
import android.content.IntentFilter;

import android.content.BroadcastReceiver;
import android.content.SharedPreferences;
import android.location.Address;
import android.location.Geocoder;
import android.location.Location;
import android.location.LocationManager;

import android.os.Bundle;
import android.telephony.SmsManager;
import android.telephony.SmsMessage;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.CheckBox;
import android.widget.ListView;

public class EmergencyResponder extends Activity {

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState) ;
    setContentView(R.layout.main) ;
  }
}
```

2. Add permissions for finding your location as well as sending and receiving incoming SMS messages to the project manifest.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest   xmlns:android="http://schemas.android.com/apk/res/
android "
    package="com.paad.emergencyresponder" >
<application
android:icon="@drawable/icon"
android:label="@string/app_name">
<activity
```

```
android:name=".EmergencyResponder "
android:label="@string/app_name">
<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
</application>
<uses-permission android:name="android.permission.RECEIVE_SMS"/>
<uses-permission android:name="android.permission.SEND_SMS"/>
<uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION"
    />
</manifest>
```

3.Modify the main.xml layout resource. Include a Listview to display the list of people requesting a status update and a series of buttons for sending response SMS messages. Use external resource references to fill in the button text; you'll create them in Step 4.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="fill_parent"
android:layout_height="fill_parent">
<TextView
android:id="@+id/labelRequestList"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="people want to know if you're ok"
android:layout_alignParentTop="true"
    />
<LinearLayout
android:id="@+id/buttonLayout"
    xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:padding="5px"
```

```
android:layout_alignParentBottom="true" >
<CheckBox
android:id="@+id/checkboxSendLocation"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="Include Location in Reply"/>
<Button
android:id="@+id/okButton"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="@string/respondAllClearButtonText"/>
<Button
android:id="@+id/notOkButton"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="@string/respondMaydayButtonText"/>
<Button
android:id="@+id/autoResponder"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="Setup Auto Responder"/>
</LinearLayout>
<ListView
android:id="@+id/myListView"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:layout_below="@id/labelRequestList"
android:layout_above="@id/buttonLayout"/>
</RelativeLayout>
```

4. Update the external strings.xml resource to include the text for each button and default response messages to use when responding, including "I'm safe" or "I'm in danger" messages. You should also define the incoming message text to use when your phone detects requests for status responses.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<string name="app_name">Emergency Responder</string>
<string name="respondAllClearButtonText">I am Safe and Well
```

```
</string>
<string name="respondMaydayButtonText">MAYDAY! MAYDAY! MAYDAY!
</string>
<string name="respondAllClearText">I am safe and well. Worry not!
</string>
<string name="respondMaydayText">Tell my mother I love her.
</string>
<string name="querystring">are you ok?</string>
</resources>
```

At this point, the GUI will be complete, so starting the application should show you the following screen:

Emergency Responder app

5. Create a new ArrayList of Strings within the EmergencyResponder Activity to store the phone numbers of the incoming requests for your status. Bind the ArrayList to the List View, using an Array Adapter in the Activity's onCreate method, and create a new ReentrantLock object to ensure thread-safe handling of the ArrayList.
6. Take the opportunity to get a reference to the CheckBox and to add ClickListeners for each of the response buttons. Each button should call the respond method, while the Setup Auto Responder button should call the startAutoResponder stub.

```
ReentrantLock lock;
 CheckBox locationCheckBox;
 ArrayList<String> requesters;
 ArrayAdapter<String> aa;

 @Override
 public void onCreate(Bundle savedInstanceState){
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
```

```
    lock = new ReentrantLock();
    requesters = new ArrayList<String>();
    wireUpControls();
 }

 private void wireUpControls() {
             locationCheckBox    =    (CheckBox)findViewById(R.
id.checkboxSendLocation);
   ListView myListView = (ListView)findViewById(R.id.myListView);

    int layoutID = android.R.layout.simple_list_item_1;
    aa = new ArrayAdapter<String>(this, layoutID, requesters);
    myListView.setAdapter(aa);

    Button okButton = (Button)findViewById(R.id.okButton);
    okButton.setOnClickListener(new OnClickListener() {
    public void onClick(View arg0) {
      respond(true, locationCheckBox.isChecked());
    }
    });

    Button notOkButton = (Button)findViewById(R.id.notOkButton);
    notOkButton.setOnClickListener(new OnClickListener() {
    public void onClick(View arg0) {

      respond(false, locationCheckBox.isChecked());
    }
    });

    Button autoResponderButton =
    (Button)findViewById(R.id.autoResponder);
    autoResponderButton.setOnClickListener(new OnClickListener()
{
    public void onClick(View arg0) {
      startAutoResponder();
    }
    });
}
```

```
public void respond(boolean _ok, boolean _includeLocation) {}
private void startAutoResponder() {}
```

7. Next, implement a Broadcast Receiver that will listen for incoming SMS messages.

a. Start by creating a new static string variable to store the incoming SMS message intent action.

```
public static final String SMS_RECEIVED =
  "android.provider.Telephony.SMS_RECEIVED";
```

b. Then create a new Broadcast Receiver as a variable in the EmergencyResponder Activity. The receiver should listen for incoming SMS messages and call the requestReceived method when it sees SMS messages containing the "are you safe" String you defined as an external resource in Step 4.

```
BroadcastReceiver emergencyResponseRequestReceiver =
    new BroadcastReceiver() {
    @Override
    public void onReceive(Context _context, Intent _intent) {
      if (_intent.getAction().equals(SMS_RECEIVED)) {
        String queryString = getString(R.string.querystring);

        Bundle bundle = _intent.getExtras();
        if (bundle != null) {
          Object[] pdus = (Object[]) bundle.get("pdus");
          SmsMessage[] messages = new SmsMessage[pdus.length];
          for (int i = 0; i < pdus.length; i++)
            messages[i] =
             SmsMessage.createFromPdu((byte[]) pdus[i]);

          for (SmsMessage message : messages) {
            if (message.getMessageBody().toLowerCase().contains
               (queryString))
             requestReceived(message.getOriginatingAddress());
          }
        }
      }
```

```
    }
  };

public void requestReceived(String _from) {}
```

8. Update the onCreate method of the Emergency Responder Activity to register the Broadcast Receiver created in Step 7.

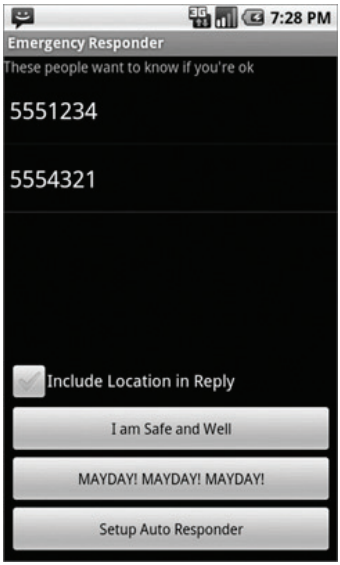
```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  lock = new ReentrantLock();
  requesters = new ArrayList<String>();
  wireUpControls();

IntentFilter filter = new IntentFilter(SMS_RECEIVED);
  registerReceiver(emergencyResponseRequestReceiver, filter);
 }
```

9. Update the requestReceived method stub so that it adds the originating number of each status request's SMS to the "requesters" ArrayList.

```
public void requestReceived(String _from) {
if (!requesters.contains(_from)) {
  lock.lock();
  requesters.add(_from);
  aa.notifyDataSetChanged();
  lock.unlock();
  }
}
```

10. The Emergency Responder Activity should now be listening for status request SMS messages and adding them to the ListView as they arrive. Start the application and send SMS messages to the device or emulator on which it's running.
11. Now update the Activity to let users respond to these status requests. Start by completing the respond method stub you created in Step 6. It should iterate over the ArrayList of status requesters and send a new SMS message to each. The SMS message text should be based on

the response strings you defined as resources in Step 4. Fire the SMS using an overloaded respond method that you'll complete in the next step.



You can key in emergency numbers too

```
  public void respond(boolean _ok,
boolean _includeLocation) {
  String okString = getString(R.
string.respondAllClearText);
    String notOkString = getString(R.
string.respondMaydayText);
    String outString = _ok ? okString
: notOkString;

    ArrayList<String> requestersCopy
=
      (ArrayList<String>)requesters.
clone();

    for (String to : requestersCopy)
      respond(to, outString, _includeLocation);
  }

  private void respond(String _to, String _response,
                       boolean _includeLocation) {}
```

12. Update the respond method that handles the sending of each response SMS. Start by removing each potential recipient from the "requesters" ArrayList before sending the SMS. If you're responding with your current location, use the Location Manager to find it before sending a second SMS with your current position as raw longitude/latitude points and a geo-coded address.

```
  public void respond(String _to, String _response,
                      boolean _includeLocation) {
  // Remove the target from the list of people we
      // need to respond to.
      lock.lock();
      requesters.remove(_to);
```

```
aa.notifyDataSetChanged();
lock.unlock();

SmsManager sms = SmsManager.getDefault();

// Send the message
sms.sendTextMessage(_to, null, _response, null, null);

StringBuilder sb = new StringBuilder();

// Find the current location and send it
// as SMS messages if required.
if (_includeLocation) {
 String ls = Context.LOCATION_SERVICE;
LocationManager lm = (LocationManager)getSystemService(ls);
 Location l =
   lm.getLastKnownLocation(LocationManager.GPS_PROVIDER);

sb.append("I'm @:\n");
sb.append(l.toString() + "\n");

List<Address> addresses;
Geocoder g = new Geocoder(getApplicationContext(),
                          Locale.getDefault());
try {
  addresses = g.getFromLocation(l.getLatitude(),
                                l.getLongitude(), 1);
  if (addresses != null) {

  Address currentAddress = addresses.get(0);
  if (currentAddress.getMaxAddressLineIndex() > 0) {
    for (int i = 0;
         i < currentAddress.getMaxAddressLineIndex();
         i++)
    {
      sb.append(currentAddress.getAddressLine(i));
      sb.append("\n");
    }
```

```
        }
        else {
          if (currentAddress.getPostalCode() != null)
            sb.append(currentAddress.getPostalCode());
        }
      }
      } catch (IOException e) {}

      ArrayList<String> locationMsgs =
      sms.divideMessage(sb.toString());
      for (String locationMsg : locationMsgs)
      sms.sendTextMessage(_to, null, locationMsg, null, null);
    }
  }
```

13. In emergencies, it's important that messages get through. Improve the robustness of the application by including auto-retry functionality. Monitor the success of your SMS transmissions so that you can rebroadcast a message if it doesn't successfully send.

a. Start by creating a new public static String in the Emergency Responder Activity to be used as a local "SMS Sent" action.

```
public static final String SENT_SMS =
    "com.paad.emergencyresponder.SMS_SENT";
```

b. Update the respond method to include a new PendingIntent that broadcasts the action created in the previous step when the SMS transmission has completed. The packaged Intent should include the intended recipient's number as an extra.

```
public void respond(String _to, String _response,
                    boolean _includeLocation) {
    // Remove the target from the list of people we
    // need to respond to.
    lock.lock();
    requesters.remove(_to);
    aa.notifyDataSetChanged();
    lock.unlock();

    SmsManager sms = SmsManager.getDefault();

    Intent intent = new Intent(SENT_SMS);
```

```
        intent.putExtra("recipient", _to);

        PendingIntent sent =
        PendingIntent.getBroadcast(getApplicationContext(),
                                   0, intent, 0);

        // Send the message
        sms.sendTextMessage(_to, null, _response, sent, null);

        StringBuilder sb = new StringBuilder();

        if (_includeLocation) {
       [ ... existing respond method that finds the location ... ]
        ArrayList<String> locationMsgs =
          sms.divideMessage(sb.toString());
        for (String locationMsg : locationMsgs)
          sms.sendTextMessage(_to, null, locationMsg, sentIntent,
null);
        }
    }
```

c. Then implement a new Broadcast Receiver to listen for this Broadcast Intent. Override its onReceive handler to confirm that the SMS was successfully delivered; if it wasn't, then put the intended recipient back onto the requester ArrayList.

```
    private BroadcastReceiver attemptedDeliveryReceiver = new
    BroadcastReceiver() {
      @Override
      public void onReceive(Context _context, Intent _intent) {
      if (_intent.getAction().equals(SENT_SMS)) {
        if (getResultCode() != Activity.RESULT_OK) {
         String recipient = _intent.getStringExtra("recipient");
          requestReceived(recipient);
        }
      }
     }
   };
```

d. Finally, register the new Broadcast Receiver by extending the onCreate

method of the Emergency Responder Activity.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    lock = new ReentrantLock();
    requesters = new ArrayList<String>();
    wireUpControls();

    IntentFilter filter = new IntentFilter(SMS_RECEIVED);
   registerReceiver(emergencyResponseRequestReceiver, filter);
  IntentFilter attemptedDeliveryfilter = new IntentFilter(SENT_
SMS);
    registerReceiver(attemptedDeliveryReceiver,
                     attemptedDeliveryfilter);
}
```

7.5 Automating the Emergency Responder

In the following example, you'll fill in the code behind the Setup Auto Responder button added in the previous example, to let the Emergency Responder automatically respond to status update requests.

1. Start by creating a new autoresponder.xml layout resource that will be used to lay out the automatic response configuration window. Include an EditText for entering a status message to send, a Spinner for choosing the auto-response expiry time, and a CheckBox to let users decide whether they want to include their location in the automated responses.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent">
<TextView
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="Respond With"/>
```

```
<EditText
android:id="@+id/responseText"
android:layout_width="fill_parent"
android:layout_height="wrap_content"/>
<CheckBox
android:id="@+id/checkboxLocation"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="Transmit Location" />
<TextView
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="Auto Respond For"/>
<Spinner
android:id="@+id/spinnerRespondFor"
android:layout_width="fill_parent"

android:layout_height="wrap_content"
android:drawSelectorOnTop="true" />
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="horizontal"
android:layout_width="fill_parent"
android:layout_height="wrap_content">
<Button
android:id="@+id/okButton"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Enable"/>
<Button
android:id="@+id/cancelButton"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Disable"/>
</LinearLayout>
</LinearLayout>
```

2. Update the application's string.xml resource to define a name for an application SharedPreference and strings to use for each of its keys.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<string name="app_name">Emergency Responder</string>
<string name="respondAllClearButtonText">I am Safe and Well
</string>
<string name="respondMaydayButtonText">MAYDAY! MAYDAY! MAYDAY!
</string>
<string name="respondAllClearText" >I am safe and well. Worry
not!
</string>
<string name="respondMaydayText">Tell my mother I love her.
</string>
<string name="querystring">"are you ok?"</string>

<string
            name="user_preferences">com.paad.emergencyresponder.
preferences
</string>
<string name="includeLocationPref">PREF_INCLUDE_LOC</string>
<string name="responseTextPref">PREF_RESPONSE_TEXT</string>
<string name="autoRespondPref">PREF_AUTO_RESPOND</string>
<string name="respondForPref">PREF_RESPOND_FOR</string>
</resources>
```

3. Then create a new arrays.xml resource, and create arrays to use for populating the Spinner.

```
<resources>
<string-array name="respondForDisplayItems">
<item>- Disabled -</item>
<item>Next 5 minutes</item>
<item>Next 15 minutes</item>
<item>Next 30 minutes</item>
<item>Next hour</item>
<item>Next 2 hours</item>
<item>Next 8 hours</item>
</string-array>

<array name="respondForValues">
<item>0</item>
```

```
<item>5</item>
<item>15</item>
<item>30</item>
<item>60</item>
<item>120</item>
<item>480</item>
</array>
</resources>
```

4. Now create a new AutoResponder Activity, populating it with the layout you created in Step 1.

```
package com.paad.emergencyresponder;

import android.app.Activity;
import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.res.Resources;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.content.BroadcastReceiver;
import android.content.SharedPreferences;
import android.content.SharedPreferences.Editor;
import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.CheckBox;
import android.widget.EditText;
import android.widget.Spinner;

public class AutoResponder extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.autoresponder);
  }
 }
```

5. Update onCreate further to get references to each of the controls in the

layout and wire up the Spinner using the arrays defined in Step 3. Create two new stub methods savePreferences and updateUIFromPreferences that will be updated to save the autoresponder settings to a named SharedPreference and apply the saved SharedPreferences to the current UI, respectively.

```
Spinner respondForSpinner;
CheckBox locationCheckbox;
EditText responseTextBox;
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.autoresponder);
```

a) Start by getting references to each View.

```
            respondForSpinner    =    (Spinner)findViewById(R.
id.spinnerRespondFor);
            locationCheckbox    =    (CheckBox)findViewById(R.
id.checkboxLocation);
    responseTextBox = (EditText)findViewById(R.id.responseText);
```

b) Populate the Spinner to let users select the auto-responder expiry time.

```
ArrayAdapter<CharSequence> adapter =
    ArrayAdapter.createFromResource(this,
      R.array.respondForDisplayItems,
      android.R.layout.simple_spinner_item);

  adapter.setDropDownViewResource(
    android.R.layout.simple_spinner_dropdown_item);
  respondForSpinner.setAdapter(adapter);
```

c) Now wire up the OK and Cancel buttons to let users save or cancel setting changes.

```
Button okButton = (Button) findViewById(R.id.okButton);
okButton.setOnClickListener(new View.OnClickListener() {
  public void onClick(View view) {
```

```
            savePreferences();
            setResult(RESULT_OK, null);
            finish();
        }
    });

    Button cancelButton = (Button) findViewById(R.id.cancelButton);
    cancelButton.setOnClickListener(new View.OnClickListener() {
        public void onClick(View view) {
            respondForSpinner.setSelection(-1);
            savePreferences();
            setResult(RESULT_CANCELED, null);
            finish();
        }
    });
```

d) Finally, make sure that when the Activity starts, it updates the GUI to represent the current settings.

```
    // Load the saved preferences and update the UI
    updateUIFromPreferences();
```

e) Close off the onCreate method, and add the updateUIFromPreferences and savePreferences stubs.

```
    }

    private void updateUIFromPreferences() {}
    private void savePreferences() {}
```

6. Next, complete the two stub methods from Step 5. Start with updateUIFromPreferences; it should read the current saved AutoResponderpreferences and apply them to the UI.

```
    private void updateUIFromPreferences() {
        // Get the saves settings
            String preferenceName = getString(R.string.user_
preferences);
```

```
        SharedPreferences sp = getSharedPreferences(preferenceNa
me, 0);

    String autoResponsePref = getString(R.string.autoRespondPref);
            String responseTextPref = getString(R.string.
responseTextPref);
     String autoLocPref = getString(R.string.includeLocationPref);
      String respondForPref = getString(R.string.respondForPref);

         boolean autoRespond = sp.getBoolean(autoResponsePref,
false);
       String respondText = sp.getString(responseTextPref, "");
       boolean includeLoc = sp.getBoolean(includeLocPref, false);
       int respondForIndex = sp.getInt(respondForPref, 0);

       // Apply the saved settings to the UI
       if (autoRespond)
       respondForSpinner.setSelection(respondForIndex);
       else
       respondForSpinner.setSelection(0);

       locationCheckbox.setChecked(includeLoc);
       responseTextBox.setText(respondText);
   }
```

7. Complete the savePreferences stub to save the current UI settings to a Shared Preferences file.

```
   private void savePreferences() {
       // Get the current settings from the UI
       boolean autoRespond =
       respondForSpinner.getSelectedItemPosition() > 0;
               int respondForIndex = respondForSpinner.
getSelectedItemPosition();
       boolean includeLoc = locationCheckbox.isChecked();
       String respondText = responseTextBox.getText().toString();

       // Save them to the Shared Preference file
            String preferenceName = getString(R.string.user_
```

```
preferences);
        SharedPreferences sp = getSharedPreferences(preferenceNa
me, 0);

       Editor editor = sp.edit();
       editor.putBoolean(getString(R.string.autoRespondPref),
                         autoRespond);
       editor.putString(getString(R.string.responseTextPref),
                                    respondText);
       editor.putBoolean(getString(R.string.includeLocationPref),
                                     includeLoc );
        editor.putInt(getString(R.string.respondForPref),respondF
orIndex);
       editor.commit();

       // Set the alarm to turn off the autoresponder
       setAlarm(respondForIndex);
    }

    private void setAlarm(int respondForIndex) {}
```

8. The setAlarm stub from Step 7 is used to create a new Alarm that fires an Intent that should result in the AutoResponder being disabled.
   You'll need to create a new Alarm object and a BroadcastReceiver that listens for it before disabling the auto-responder accordingly.

a) Start by creating the action String that will represent the Alarm Intent.

```
public static final String alarmAction =
"com.paad.emergencyresponder.AUTO_RESPONSE_EXPIRED";
```

b) Then create a new Broadcast Receiver instance that listens for an Intent that includes the action specified in Step 8.a When this Intent is received, it should modify the auto-responder settings to disable the automatic response.

```
private BroadcastReceiver stopAutoResponderReceiver = new
BroadcastReceiver() {
  @Override
```

```
        public void onReceive(Context context, Intent intent) {
        if (intent.getAction().equals(alarmAction)) {
                String preferenceName = getString(R.string.user_
preferences);
           SharedPreferences sp = getSharedPreferences(preference
Name,0);

          Editor editor = sp.edit();
            editor.putBoolean(getString(R.string.autoRespondPref),
false);
          editor.commit();
        }
       }
    };
```

c) Finally, complete the setAlarm method. It should cancel the existing alarm if the auto-responder is turned off; otherwise, it should update the alarm with the latest expiry time.

```
    PendingIntent intentToFire;

    private void setAlarm(int respondForIndex) {
      // Create the alarm and register the alarm intent receiver.

      AlarmManager alarms =
        (AlarmManager)getSystemService(ALARM_SERVICE);

      if (intentToFire == null) {
        Intent intent = new Intent(alarmAction);
        intentToFire =
          PendingIntent.getBroadcast(getApplicationContext(),
                                     0,intent,0);

        IntentFilter filter = new IntentFilter(alarmAction);

        registerReceiver(stopAutoResponderReceiver, filter);
       }
```

```
    if (respondForIndex < 1)
    // If "disabled"  is selected, cancel the alarm.
    alarms.cancel(intentToFire);
    else {
      // Otherwise find the length of time represented
      // by the selection and and set the alarm to
      // trigger after that time has passed.
      Resources r = getResources();
      int[] respondForValues =
        r.getIntArray(R.array.respondForValues);
      int respondFor = respondForValues [respondForIndex];

      long t = System.currentTimeMillis();
      t = t + respondFor*1000*60;

      // Set the alarm.
      alarms.set(AlarmManager.RTC_WAKEUP, t, intentToFire);
    }
  }
```

9. That completes the AutoResponder, but before you can use it, you'll need to add it to your application manifest.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.paad.emergencyresponder">
<application
android:icon="@drawable/icon"
android:label="@string/app_name">
<activity
android:name=".EmergencyResponder"
android:label="@string/app_name">
<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER"  />
</intent-filter>
</activity>
```

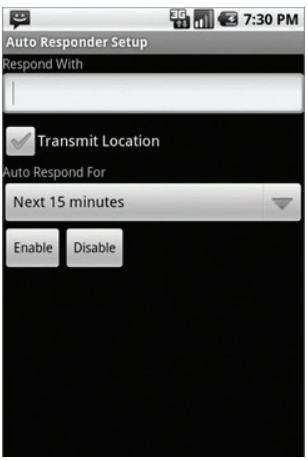
```
  <activity
  android:name=".AutoResponder"
  android:label="Auto Responder Setup"/>
  </application>
  <uses-permission   android:name="android.permission.ACCESS_
GPS"/>
  <uses-permission
  android:name="android.permission.ACCESS_LOCATION"/>
  <uses-permission   android:name="android.permission.RECEIVE_
SMS"/>
  <uses-permission android:name="android.permission.SEND_SMS"/>
  </manifest>
```

To enable the auto-responder, return to the Emergency Responder Activity and update the startAutoResponder method stub that you created in the previous example. It should open the AutoResponder Activity you just created.

```
  private void startAutoResponder() {
  startActivityForResult(new Intent(EmergencyResponder.this,
                                    AutoResponder.class), 0);
  }
```

10. If you start your project, you should now be able to bring up the Auto Responder Setup window to set the auto-response settings.
11. The final step is to update the request Received method in the Emergency Responder Activity to check if the auto-responder has been enabled. If it has, the request Received method should automatically execute the respond method, using the message and location settings defined in the application's Shared Preferences.

```
  public void requestReceived(String
_from) {
    if (!requesters.contains(_from)) {
      lock.lock();
      requesters.add(_from);
```



Autoresponder

```
        aa.notifyDataSetChanged();
        lock.unlock();
    // Check for auto-responder
            String preferenceName = getString(R.string.user_
preferences);
        SharedPreferences prefs = getSharedPreferences(preferenc
eName,
                                                        0);
      String autoRespondPref = getString(R.string.autoRespondPref)
         boolean autoRespond = prefs.getBoolean(autoRespondPref,
false);
        if (autoRespond) {
          String responseTextPref =
            getString(R.string.responseTextPref);
          String includeLocationPref =
            getString(R.string.includeLocationPref);

           String respondText = prefs.getString(responseTextPref,
"");
        boolean includeLoc = prefs.getBoolean(includeLocationPref,
                                              false);

          respond(_from, respondText, includeLoc);
        }
      }
    }
```

You should now have a fully functional interactive and automated emergency responder.

The telephony stack is one of the fundamental technologies available on mobile phones. While not all Android devices will necessarily provide telephony APIs, those that do are particularly versatile platforms for person-to-person communication.

8 Wi-Fi

The wifiManager represents the Android Wi-Fi Connectivity Service. It can be used to configure Wi-Fi network connections, manage the current Wi-Fi connection, scan for access points and monitor changes in Wi-Fi connectivity.

As with the Connectivity Manager, you access the Wi-Fi Manager using the getsystemservice method, passing in the Context.WIFI_SERVICE constant-

```
String service = Context.WIFI_SERVICE;
WifiManager wifi = (WifiManager)getSystemService(service);
```

To use the Wi-Fi Manager, your application must have uses-permissions for accessing and changing the Wi-Fi state included in its manifest.

```
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
```

You can use the Wi-Fi Manager to enable or disable your Wi-Fi hardware using the setwifiEnabled method, or request the current Wi-Fi state using the getwifistate or isWifiEnabled methods:

```
if (!wifi.isWifiEnabledf))
      if (wifi.getWifiState() != WifiManager.WIFI_STATE_ENABLING)
         wifi.setwifiEnabled(true);
```

The WifiManager also provides low-level access to the Wi-Fi network configurations. You have full control over each Wi-Fi configuration setting, which enables you to completely replace the native Wi-Fi management application if required. Later in this section you'll get a brief introduction to the APIs used to create, delete and modify network configurations.

8.1 Monitoring Wi-Fi Connectivity

The Wi-Fi Manager broadcasts Intents whenever the connectivity status of the Wi-Fi network changes, using an action from one of the following constants defined in the WifiManager class:

- **WIFI_STATE_CHANGED_ACTION:** Indicates that the Wi-Fi hardware status has changed, moving between enabling, enabled, disabling, disabled and unknown. It includes two extra values keyed on EXTRA_WIFI_STATE and EXTRA_PREVIOUS_STATE that provide the new and previous Wi-Fi states, respectively.
- **SUPPLICANT_CONNECTION_CHANGE_ACTION:** This Intent is broadcast whenever the connection state with the active supplicant

(access point) changes. It's fired when a new connection is established or an existing connection is lost, using the EXTRA_NEW_STATE Boolean extra, which returns true in the former case.

- **NETWORK_STATE_CHANGED_ACTION:** Fired whenever the Wi-Fi connectivity state changes. This Intent includes two extras: the first EXTRA_NETWORK_INFO includes a NetworkInfo object that details the current network state, while the second EXTRA_BSSID includes the BSSID (Basic Service Set IDentifier) of the access point you're connected to.
- **RSSI_CHANGED_ACTION:** You can monitor the current signal strength of the connected Wi-Fi network by listening for the RSSI_CHANGED_ACTION Intent. This Broadcast Intent includes an integer extra, EXTRA_NEW_RSSI, that holds the current signal strength. To use this signal strength, use the calculateSignalLevel static method on the Wi-Fi Manager to convert it to an integer value on a scale you specify..

8.2 Monitoring Active Connection Details

Once an active network connection has been established, use the getconnectionInfo method on the Wi-Fi Manager to find information on the active connection's status. The returned wifiInfo object includes the SSID (Service Set Identifier), BSSID, Mac address and IP address of the current access point, as well as the current link speed and signal strength.

```
WifiInfo info = wifi.getconnectionInfo();
      if (info.getBSSID()!= null) {
           int strength = WifiManager.calculateSignalLevel(info.
getRssi(), 5);
        int speed = info.getLinkSpeed();
        String units = WifiInfo.LINK_SPEED_UNITS;
        String ssid = info.getSSID();

       String cSummary = String.format("Connected to %s at %s%s.
Strength %s/5",
                                                ssid, speed, units,
strength);
      }
```

8.3 Scanning for Hotspots

You can also use the Wi-Fi Manager to conduct access point scans using the startScan method. An Intent with the SCAN_RESULTS_AVAILABLE_

ACTION action will be broadcast to asynchronously announce that the scan is complete and results are available.

Call getScanResults to get those results as a list of ScanResult objects.

Each Scan Result includes the details retrieved for each access point detected, including link speed, signal strength, SSID and the authentication techniques supported.

To initiate a scan for access points that display a Toast indicating the total number of access points found and the name of the access point with the strongest signal:

```
// Register a broadcast receiver that listens for scan results.
      registerReceiver(new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
          List<ScanResult> results = wifi.getScanResults();
          ScanResult bestSignal = null;
          for (ScanResult result : results) {
            if (bestsignal == null ||
                      WifiManager.compareSignalLevel(bestSignal.
level,result.level)<0)
              bestSignal = result;
         }

        String toastText = String.format("%s networks found. %s is
                                              the strongest.",
                                                  results.size(),
bestsignal.SSID);
       Toast.makeText(getApplicationContext(), toastText, Toast.
LENGTH_LONG);
      }
  }, new IntentFilter(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION));
      // Initiate a scan.
wifi.startScan();
```

8.4 Managing Wi-Fi Configurations

You can use the Wi-Fi Manager to manage the configured network settings and control which networks to connect to. Once connected, you can interrogate the active network connection to get additional details of its configuration and settings.

Get a list of the current network configurations using getConfiguredNetworks. The list of wifiConfiguration objects returned includes the network ID, SSID and other details for each configuration.

To use a particular network configuration, use the enableNetwork method, passing in the network ID to use and specifying true for the disableAllothers parameter:

```
// Get a list of available configurations
            List<WifiConfiguration>  configurations  =  wifi.
getConfiguredNetworks();
     // Get the network ID for the first one.
     if (configurations.size() > 0) {
       int netID = configurations.get(0).networkId;
       // Enable that network.
boolean disableAllOthers = true;
wifi.enableNetwork(netID, disableAllOtherstrue);
     }
```

8.5 Automating the configuration

To connect to a Wi-Fi network, you need to create and register a configuration. Normally, your users would do this using the native Wi-Fi configuration settings, but there's no reason you can't expose the same functionality within your own applications, or for that matter replace the native Wi-Fi configuration Activity entirely.

Network configurations are stored as Wificonfiguration objects. The following is a non-exhaustive list of some of the public fields available for each Wi-Fi configuration:

- **BSSID:** The BSSID for an access point
- **SSID:** The SSID for a particular network
- **networkId:** A unique identifier used to identify this network configuration on the current device
- **priority:** The network configuration's priority to use when ordering the list of potential access points to connect to
- **status:** The current status of this network connection, which will be one of the following: WifiConfiguration.Status.ENABLED, WifiConfiguration.Status.DISABLED or WifiConfiguration.Status.CURRENT

The configuration object also contains the supported authentication techniques, as well as the keys used previously to authenticate with this access point.

All this and more in the
world of Technology

VISIT NOW

www.thinkdigit.com

RNI REGN. No. MAH ENG/2001/3791